Reuse of Retired Mobile Devices in Cyber-Physical Systems

By

TIM AMBROSE

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

December 12, 2023

Abstract

Cyber-physical systems continue to be increasingly important to global society because of automation, miniaturization of electronics, and the Internet of Things. More smartphones and tablets are being manufactured and discarded or recycled every year. This research presents a design and characterization of a system that gives retired mobile devices further use within cyber-physical systems beyond their typical end-of-life. The contributions include an experimental characterization of wireless communication protocols for embedded systems involving wide-area networks using query vehicles, a design of an app-based distributed computing architecture named Cluster made from retired smartphones and tablets, implementation of four distributed computing tasks, and a characterization of the performance, energy consumption, and network utilization of Cluster and each of the four compute tasks. This research analyzes the potential and performance of retired, but still functional, mobile devices to be used as nodes in a distributed computer for applications such as image pre-processing, dependent parallel calculations, server-heavy coordinated computing efforts, and a distributed data store based on a key-value data store implemented on a "Fast Array of Wimpy Nodes". The results of this research demonstrate that Cluster is a viable distributed computer with some expected reliability concerns from reusing retired computing resources, comparable compute performance, lower power, greater physical space efficiency, and lower monetary costs than most other alternative distributed computing architectures.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Mobile devices have transformed many aspects of society by placing tremendous computational and communication power in the hands and pockets of billions of people. Despite decades of exponential growth, mobile devices are projected to continue increasing in number and sophistication. Figure 1.1 shows that by the year 2025 the number of mobile devices worldwide is expected to increase by 30% compared to the year 2020 to over 18 billion devices [1]. Figure 1.2 shows that the number of mobile network subscriptions is rapidly approaching the population of the planet [2]. All of these devices represent a truly vast pool of computational resources distributed across the globe.

1.1 Motivation

The growth of mobile devices and the fast pace of innovation in their capabaility has a downside, namely the rapid retirement and disposal of electronics that are likely still functional. In 2013, Americans replaced their mobile phones every 22 months [6]. In their more recent study, Cordella et al. [7] assert that most smartphones are retired after only 2 years of use, although this can be extended to 4.5 – 5.6 years if the phone is passed to a second consumer who then retires the device. Even an extended lifetime would result in millions of phones being retired each year. The production and usage of mobile devices has many impacts on sustainability, of which two primary ones are (1) contributions to global warming potential (GWP) through greenhouse gas (GHG) emissions (also known as the carbon footprint or CF) and (2) use of hazardous materials that affect the environment and human health.

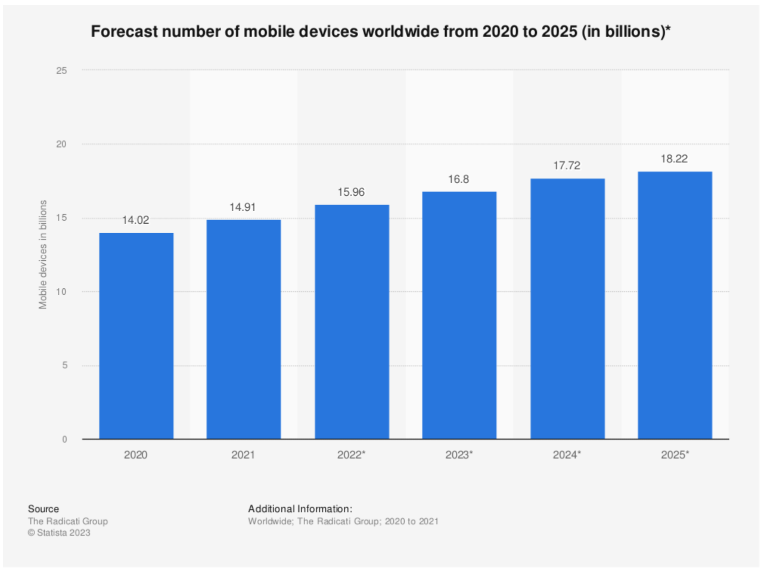


Figure 1.1: Near-term forecast number of mobile devices worldwide [1].

In their life cycle analysis, Ercan et al. [8] calculate the GWP of a smartphone to be 57 kg $CO_{2,eq}$ over a three year usage cycle, dominated by the production of the smartphone, which is in turn dominated by the production of the integrated circuits (ICs). This result is consistent with other studies that show the energy of IC production often greatly exceeds the energy used by the IC over its lifetime [9]. Cordella et al. report a range of 26.7 – 70.4 kg $CO_{2,eq}$ for the manufacturing of one device. Both studies note that the carbon footprint of networking equipment, data centers, and other infrastructure supporting smartphone services also contribute significantly to the carbon footprint, however addressing this issue is outside the scope of the present work.

Discarded mobile devices contribute significantly to electronic waste (e-waste), which is projected to reach approximately 75 million metric tonnes (Figure 1.3) by the year 2030 [3]. Ilankoon et al. [10] report that only 15% – 20% of e-waste was formally collected and treated and that much of the collected waste is transferred from the developed world to the developing world, where cheap disposal facilities and lax environmental standards and laws expose the local environment and population to a range of hazardous chemicals. While some of this material is potentially recyclable, the procedures for recycling electronics also uses hazardous chemicals to reclaim precious metals while producing carbon emissions and additional chemical waste. It is estimated that if the entire device or its components are in good working condition, reuse of electronic equipment is up to 20 times more environmentally beneficial than recycling it [10].

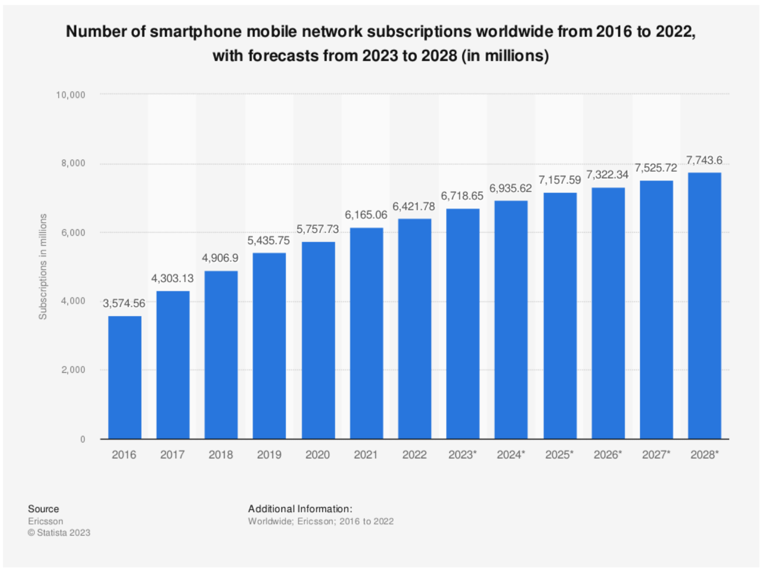


Figure 1.2: Near-term forecast number of mobile subscriptions worldwide [2].

The key insight to lessen the negative environmental impacts of mobile electronics and to make the mobile electronics economy sustainable is to displace the production of new devices elsewhere in the information and communication technology ecosystem [11]. While reusing mobile device components and subsystems has been proposed, this requires redesigning the device to be modular and upgradeable [12]. The failure of initiatives like Google's Project Ara [13] indicates that reusing the entire device, as studied in the present work, is a more viable approach.

1.2 Cyber-Physical Systems and Reused Mobile Devices

Cyber-physical systems (CPS) are systems with inputs or sensors, outputs or actuators, and some form of computation or control connecting the inputs to the outputs. The block diagram in Figure 1.4 shows the general form of a distributed cyber-physical system. Because a CPS requires a diverse set of capabilities from its components (sensing, computing, data storage, actuation), there are multiple opportunities for reusing mobile devices in a CPS. In fact, a retired mobile device could have multiple roles in a single CPS, and therefore be used repeatedly until ultimately recycled for its raw materials. For example, a new device will have a first life as a consumer information appliance, where primary tasks include interpersonal communication, social and mass media, and entertainment. Today's mobile devices typically have a range of sensors (multiple cameras, microphones, GPS, compass, accelerometer, etc.), which can support a second life

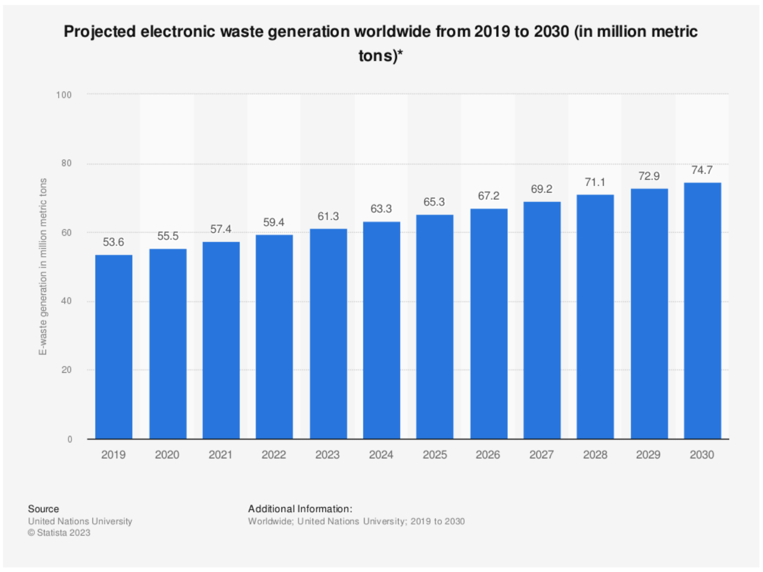


Figure 1.3: Near-term projected electronic waste worldwide [3].

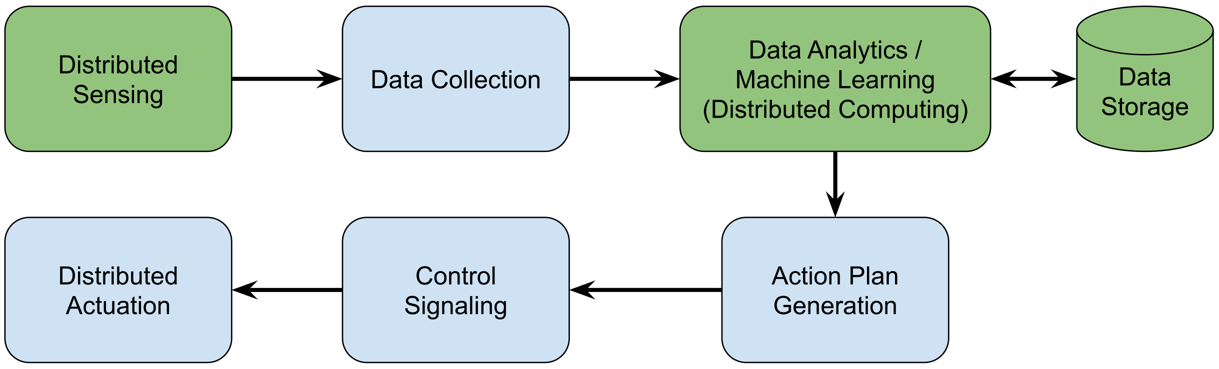


Figure 1.4: Block Diagram of a Cyber-Physical System

as a sensing node (Section 1.2.2). A mobile device's multiple communication interfaces (Wi-Fi, Bluetooth) can support a third life as an IoT gateway [14, 15] and its multiple processors could support an additional life as a compute node (see Chapter 3).

The three green blocks in Figure 1.4 (Distributed Sensing, Distributed Computing, and Data Storage) are the parts of a cyber-physical system where the present work contributes to an understanding of the role a reused mobile device can play. Distributed sensors require data collection; this may be in the form of a query vehicle (data mule) or a wireless network connecting all sensors together. Once the data is collected, it needs to be stored and analyzed. Commonly, analysis is done by machine learning for pattern recognition or data classification; such computations are often sped up through distributed or parallel computing. Once the data is analyzed, the system may decide to output the results or actuate some peripherals (e.g. motors, radios, lights). The following two sections describe scenarios in which reused mobile devices can be incorporated into a cyber-physical system.

1.2.1 Robot Arm Kinematics

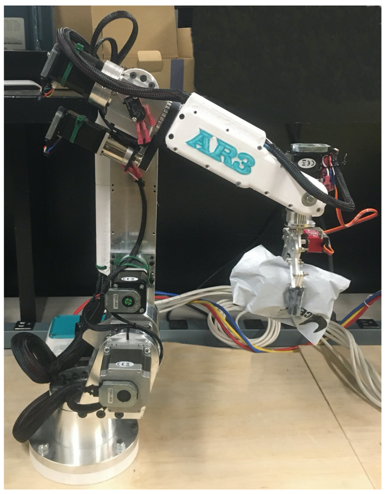


Figure 1.5: AR3 Robot Arm

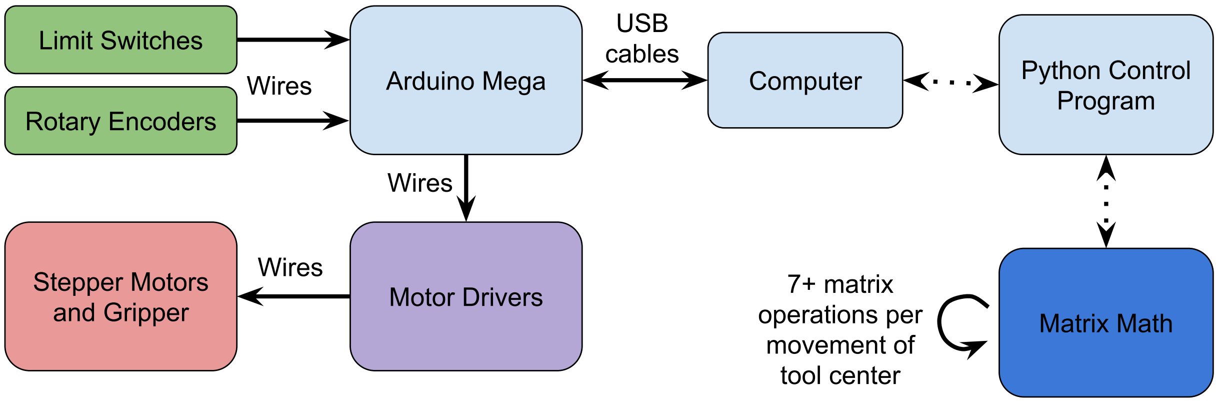


Figure 1.6: Robot Arm Operation Block Diagram

Figure 1.5 shows a commercially-available robot arm — which was assembled, programmed, and calibrated by Tim Ambrose over the course of 3 months — that is currently being integrated into research projects in agricultural robotics at UC Davis. The arm is an example of a localized (non-spatially distributed) cyber-physical system. Figure 1.6 shows a block diagram of the robot arm system. Because the arm relies on specialized sensors and actuators that interface directly to an Arduino Mega microcontroller, there is limited opportunity to reuse mobile devices in sensing and actuation. However, controlling the robot arm and moving its tool center requires at least 7 matrix operations per millimeter of movement (blue box). The software for the AR3 robot arm is free and open source and therefore could be preprogrammed to outsource it's matrix operations to a distributed computer. Some work has already been done to customize that software as part of this present research.

DroidCluster [16] demonstrated that a cluster of six LG P500 Android phones (each with a Qualcomm MSM7227 CPU containing a 600 MHz ARM11 core) was able to achieve 26 MFLOPs using a wireless LAN network and 29 MFLOPs using a USB hub to connect to a PC on the LINPACK linear algebra benchmark. The LINPACK implementation was based on the Message-Passing Interface library and required the Android operating systems on the phones to be augmented with a Debian Linux OS that runs under the same kernel. This arrangement allows the Android OS to function normally, which is a requirement for phones still be used by consumers, but is not necessary for retired phones. Despite the complexity of the implementation, the floating-point performance of the 6-node DroidCluster is more than sufficient to support the real-time computation needed to move the robot arm. Indeed, a set of several arms could plausibly be supported by the same collection of mobile phones. The DroidCluster work shows a proof-of-concept that small clusters of mobile phones can deliver sufficient computational power to support cyber-physical systems.

1.2.2 Bioacoustic Monitoring

The Rainforest Connection (https://rfcx.org/) is a non-profit organization working to preserve rainforests using bioacoustic monitoring to track illegal extractive activities such as logging, mining, and oil and gas exploration. Starting in 2016, the group installed treetop acoustic monitoring units implemented with used cellphones powered by solar panels to upload approximately 200 MB of audio data every 24 hours [17]. The audio data is processed in the cloud to detect the sound of chain saws or other equipment linked to illegal logging. The natural rainforest sounds are also analyzed to monitor different species or study biodiversity. Deichmann et al. [18] in a 2017 study used a network of 10 LG L70 cellular phones as autonomous audio recorders to study the impact of natural gas exploration on tropical forest biodiversity. Both of these are examples of spatially-distributed cyber-physical systems where used mobile phones are utilized in distributed sensing and data collection (as network gateways) roles. Other examples include the proposed reuse of smartphones as camera traps [19].

As technology evolves, retired mobile phones may be replaced by custom hardware. In the case of the Rainforest Connection's Guardian platform, the need for satellite communications connectivity drove the development of a new acoustic monitoring system to replace the phones [20]. However, this transition highlights another use case for retired mobile devices in cyber-physical systems: as prototyping platforms for distributed sensing. Because the software ecosystem of mobile devices enables rapid app development, retired devices can be repurposed primarily through software to accomplish distributed sensing or other functions while optimized, custom hardware is developed to replace it.

1.3 Distributed Computing and Reused Mobile Devices

Distributed computing refers to a very broad range of computing architectures that involve using multiple computers working together to achieve a goal. The computers could be located near each other as in a data center or distributed across the globe. The computations executing on the computers could be tightly-coupled, requiring close coordination between the computing nodes, or loosely-coupled, with each machine working largely independently of the others. Bal et al. [21] distinguish parallel computers (vector processors, dataflow machines, multiprocessors/multicore processors) from distributed computers by specifying that in a parallel computer, the processors share a common memory, while in a distributed computer, the processors exchange data by message-passing over a network. Multiple individual computers or workstations working together and communicating over a local area network (LAN) or wide area network (WAN) meets their definition of a distributed computer. For Bal et al., a distributed computer architecture where the computing nodes are physically close to each other and communication is fast and reliable is *closely-coupled*, while an architecture where the nodes are physically dispersed and communication is slow and unreliable is *loosely-coupled*.

The growth of the Internet and the emergence of the Internet of Things (IoT) has motivated the development of several variations on the workstation-WAN distributed computing architecture. *Grid computing* involves connecting organizationally-owned and managed computing resources (e.g., supercomputers or workstation clusters) to address scientific supercomputing applications, while *public-resource computing* attempts to utilize individually-owned personal computers for the same application space [22]. For example, projects supported by the Berkeley Open Infrastructure for Network Computing (BOINC) such as SETI@home used idle cycles on personal computers to advance their research [22]. Büsching et al. propose utilizing a similar approach to create ad-hoc distributed computing systems out of collections of physically co-located personal mobile phones, e.g. using all of the personal phones of passengers on a train to compute a local weather forecast [16].

Cloud computing is a computing model that enables convenient, on-demand, network-based access to a shared pool of configurable computing resources [23]. The major infrastructure component supporting cloud computing is the data center, which is typically an organization-owned and managed pool of servers, storage, networking equipment, etc., located physically together, in an architecture analogous to the workstation-LAN distributed computing model. For some IoT applications, the centralization of computing hardware at the data center can introduce significant communication delays that negatively impact latency and application performance. To address this challenge, researchers have proposed spreading computational resources through a hierarchical model of IoT computing that incorporates *fog computing*, *mist computing*, and *edge computing* [24]. Edge computing, at the lowest level of the hierarchy, utilizes the sensors or end-user devices at the edge of the IoT for computing tasks, e.g. performing data compression before the data is uploaded to the cloud. Fog computing locates some computational resources physically near the edge nodes, to reduce latency. Some of these fog nodes may be scaled-down versions of cloud computing infrastructure, or *cloudlets*. Mist computing places lightweight computational resources even closer to the edge nodes. For example, mist nodes may be microcontrollers or microcomputers that aggregate and process data from the edge devices before passing it to the more powerful fog nodes.

Several research groups have explored how retired mobile devices can be given roles in the IoT computing hierarchy. Shahrad and Wentzlaff [25] explored how deploying decommissioned mobile phones in Infrastructure-as-a-Service (IaaS) cloud data centers could affect the total cost of ownership (TCO) compared to conventional servers. Switzer at al. demonstrated that a collection of ten three-year-old smartphones could provide good performance on IaaS cloud microservice benchmarks [26]. The Renée project demonstrated that a bank of four retired phones managed by a single central computer could function as a small-scale data center (cloudlet) or fog node to provide Function-as-a-Service (FaaS)/Platform-as-a-Service (PaaS) capabilities [27]. The present work is distinct from these prior efforts and the contributions of this research are outlined in Section 1.4.

1.4 Contributions

The research presented in this dissertation contributes to the literature in the field of mobile device reuse in cyber-physical systems in the following areas:

- Experimental characterization and analysis of the most commonly used wireless interfaces available on mobile devices and embedded systems when exchanging data with a query vehicle/data mule.
- Design and development of software on multiple mobile operating systems that supports tasks typical of CPS, for example image data preprocessing and local data storage. Development of software for desktop and server computers to coordinate the efforts of the mobile devices.
- Experimental characterization of a distributed computer consisting of a cluster of heterogeneous mobile devices networked by a wireless LAN. The characterization includes performance analysis on application tasks, measurements of power consumption, and computational carbon intensity (CCI).
- Experimental characterization of application performance of the cluster when isolated from external wireless interference.

1.5 Organization of Dissertation

Chapter 1 presents the motivation for this work and an overview of prior work on reusing retired mobile devices in cyber-physical systems. Chapter 2 presents an experimental characterization of different wireless interface transactions that transfer data from mobile devices to a query vehicle or data mule. These transfers could support the distributed sensing function of a CPS while the query vehicle implements the data collection function when the sensing devices are deployed outside the range of a wireless network (Figure 1.4). Chapter 3 describes the design of Cluster, the distributed computing system implemented in the present work. Chapter 4 presents the implementation of the tasks used to characterize the performance of the cluster. Chapter 5 presents the results of the task characterization experiments. Finally, Chapter 6 presents conclusions from this work and describes future work.

1.6 Definitions

Multi-Process Program: A program that occupies multiple processes and multiple processor cores when it runs. A typical program may have multiple threads running concurrently, which allows several algorithms to execute at the same time, but such programs do not run each of these threads simultaneously. Instead the processor switches rapidly between threads, giving approximately the same effect as if the threads were executing truly simultaneously, much like a person dividing their attention to multiple tasks, one at a time. A multi-threaded program typically runs on only one process of the operating system and therefore on only one processor core. A multi-process program *does* accomplish several algorithms running simultaneously by splitting the program across multiple processes, each of which *may* occupy a different processor core, in which case those parts *will* be executed truly simultaneously with the parts running on the other processor cores.

Distributed Computer: The definition of a distributed computer in this work is similar to workstation-LAN architecture definition of Bal et al. [21], namely a collection of distinct computers working together to solve a computational problem or provide a collection of services to an outside agent.

Cluster: The primary design and characterization effort of this research, a system of retired smartphones and tablets that acts as a distributed computer. See Chapter 3.

FAWN: Fast Array of Wimpy Nodes, a network of low-power compute or storage nodes with a specific implementation, designed by Andersen et al. [28, 29] See Section 4.1.2.

Cyber-Physical System Task: A compute task designed for Cluster which is a full implementation of an example application for a distributed computer. The tasks targeted by Cluster support services typically required in cyber-physical systems. See Section 4.1.

Synthetic Task: A compute task designed for Cluster which is not an application implementation, but rather designed to simulate a compute task with specific computational or network behavior or characteristics. See Section 4.2.

Query Vehicle: Data mule vehicle (UAV or ground robot) that physically moves around an array of sensor nodes and collects data wirelessly as it passes them by. See Chapter 2.

Chapter 2

Query Vehicle Transaction Characterization

This chapter is based on a paper in preparation [30] written about the transaction characterization experiments (TC experiments) done to explore the use of mobile robot query vehicles to collect data from retired mobile devices reused as distributed sensors. This work is complementary to the Cluster research that studies how heterogeneous retired mobile devices can be reused as a distributed computer as it analyzes the best transmission protocols and radios to use in a query vehicle system. One related application is the Rainforest Connection's project to detect illegal logging in Indonesia using old cell phones [17], see Section 5.8.1. The devices in that application could have their data harvested by a query vehicle that traverses the area over which the sensors are deployed, such as the vehicles discussed in this chapter. For further discussion of that application, see Section 1.2.2.

2.1 Summary

Query vehicle or data mule sensor network designs are dependent on the maximum navigation speed of the vehicle. To minimize the total transaction time between the (on-vehicle) server and client sensor node, the experiment described in this chapter characterizes five different radios in the 2.4 GHz and 915 MHz ISM bands and corresponding transmission protocols specifically for query vehicle applications. An ESP32 client

and Raspberry Pi server were fitted with each radio one at a time and tested repeatedly in identical circumstances for all five radios. The experiment showed a consistent transmission rate for each radio scheme, a wide range of connection and setup times for each radio, and established trendlines for major characteristics such as transmission time, connection time, total transaction time, and standard deviation for each of those characteristics. Under the tested conditions, Wi-Fi has the smallest transaction times for data mule networks needing to transmit more than 30 kB per transaction and Gaussian Frequency Shift Keying (GFSK) RF is the fastest for 30 kB or less.

2.2 Definitions

Query Vehicle: Vehicle (UAV or ground robot) used as a data mule that physically moves around an array of sensor nodes and collects data wirelessly as it passes them by.

Transaction: The entire interaction between query vehicle and sensor node including wake up, initializations, connection/handshake (unless using a connectionless protocol), transmission of the data stored on sensor node to query vehicle, any acknowledgment (ACK) or completion (DONE) signals, and disconnection including teardown (unless using a connectionless protocol).

Connection Time: The time it takes to wake a sensor node and get set up for transmitting the collected data to the query vehicle, including connecting to the query vehicle if the protocol is not connectionless.

Transmission Time: The time only during the transmit phase of the transaction, not including setup/connect time and teardown/disconnect time.

Radio Scheme: One of the following protocols (and corresponding radio) used to transmit data from sensor node to query vehicle: 2.4 GHz Wi-Fi, 2.4 GHz Classic Bluetooth 3.0, 2.4 GHz GFSK RF, 915 MHz LoRa, and 2.4 GHz ZigBee.

TC Experiment: The transaction characterization experiments that this chapter is presenting.

2.3 Introduction

Arrays of sensor nodes are a common system architecture for monitoring a field, rainforest, or any large spatially-dispersed group of many similar things. Examples include fields of crops with sensors placed throughout to monitor plant health or environmental conditions, warehouses and factories with equipment or food that needs monitoring, and smart cities containing grids of outdoor sensors throughout the public areas.

A common problem in such sensor arrays is how to get the data from the sensor nodes to a central server that collects and analyzes the information. Often these arrays of sensors are far enough apart and far enough away from the central server that having the nodes in direct communication with each other becomes impractical. Data mules, such as a Linux machine running on a query vehicle (ground robot or aerial vehicle) that physically pass near the sensor nodes, are one solution to such challenges.

The experiments described in this chapter characterize five different radio modules that are typically found on wireless sensor nodes and retired mobile devices reused as sensor nodes and their corresponding protocols in an environment typical of query vehicle networks to measure their performance as it would be in the following scenario: the query vehicle wakes up the sensor node when it gets within transmission range of the node and receives the sensor data the node has stored.

There are several aspects of such query vehicle or data mule networks that are active areas of research to improve the efficiency, viability, and latency of the network. See Section 2.7 for a comparison of several papers researching data mules and how they relate to this experiment.
The main issues associated with query vehicle sensor networks are vehicle path planning, vehicle deployment (and subsequent data collection) scheduling, and latency of the sensor data from creation until upload to a central server or database. Path planning and scheduling are the topic of many papers on data mule theory. This work focuses on latency. Specifically, it focuses on minimizing the total transaction time between query vehicle and client node by characterizing five common radio schemes in an experimental setup that mimics the conditions of a real query vehicle coming into proximity of a sensor node and attempting to collect data from the node in as little time as possible.

In Section 2.4, the physical layout of the experiment, the program flow of the server and client node software, and the specifications of each radio and protocol used in the TC experiment are described. Section 2.5 shows the data from the experiment and Section 2.6 discusses the results, the radio characteristics, and the metrics that the TC experiment affects. In Section 2.6.3, an analysis of the results and the implications of them are discussed. Section 2.7 describes related research to this work, Section 2.8 shows the conclusions drawn from this work, and Section 2.9 presents future work that could be done to add to the data from this work to better inform designs involving query vehicle networks.

2.4 Experimental Setup

The transaction characterization experiment consisted of a fixed client node and fixed server under strictly controlled conditions for fair comparisons of the five radio schemes.

2.4.1 Program Flow and Physical Layout

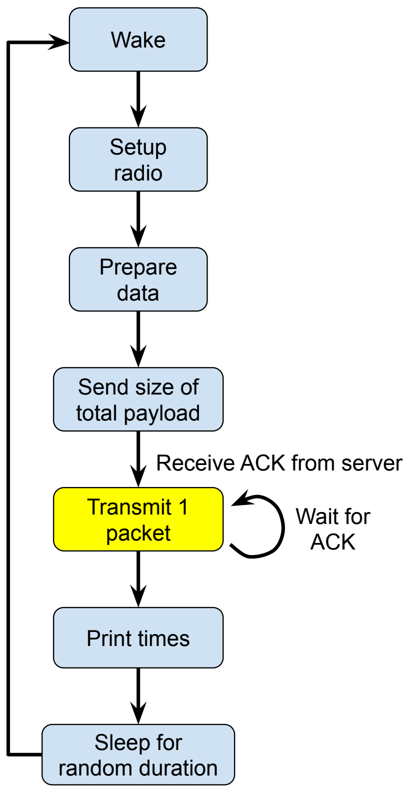


Figure 2.1: Program flow diagram for the client node software.

The client and server radios were placed 5.2 meters apart in a room with 2 cubicle walls obstructing the line-of-sight. The room area is 62 square meters. The radios were exactly the same distance apart for each tested radio scheme. The server acting as the query vehicle in this experiment was a Raspberry Pi 3 B+ running Raspian Jessie. The client acting as a node in a field of sensor nodes, was an Espressif ESP32-WROOM-32D with a 160 MHz Xtensa® 32-bit LX6 CPU (MCU).

The client node programs written (in C++) for each radio scheme of the TC experiment follow the program flow shown in Figure 2.1 with some variation in the yellow "Transmit 1 packet" step for Wi-Fi and

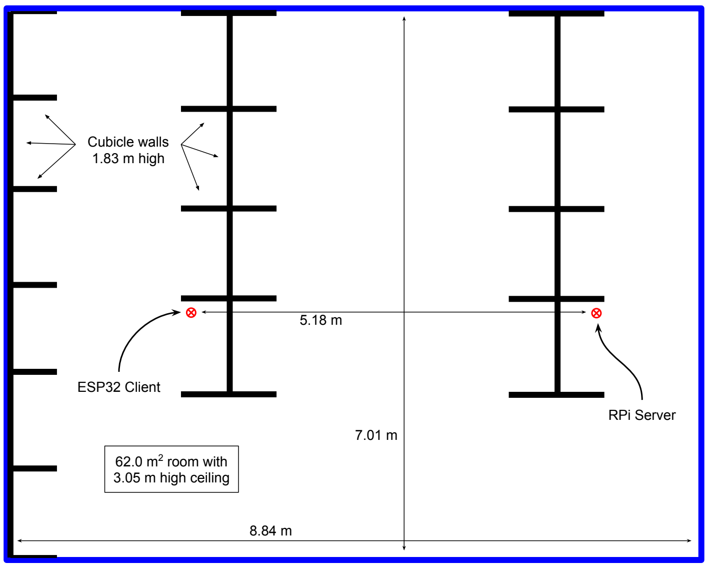


Figure 2.2: Diagram of room during TC experiment.

Bluetooth. The program assumes that the server is in range and is responsible for waking the node. First, the node wakes up from a random sleep duration by a timer interrupt. Then the radio is configured and the data is prepared for sending. A single packet goes out to the server (Raspberry Pi with matching radio) containing only an integer representing the data size in bytes. After acknowledgment that the server received that information, the data is transmitted 1 packet at a time, waiting for the ACK each time before the next one is sent, retransmitting when needed. After all packets are confirmed received, the results are recorded. The times for setup (including connection time if applicable) and transmission of the payload are timed using C library timing functions invoked on the client node.

2.4.2 Wi-Fi

The Raspberry Pi and ESP32 both have built in Wi-Fi 802.11n. The ESP32 specification is 150 Mbps for the on-board Wi-Fi. In this experiment, the Raspberry Pi broadcast a Wi-Fi network as would be needed in the field in order for the nodes to be able to connect to the query vehicle using Wi-Fi.

For Wi-Fi, TCP sockets were opened and the data was sent as fast as the TCP stack would allow. The TCP implementation inside the ESP32-WROOM-32D chip handled the selective reject and windowing protocol so the ACK packets for every transmitted payload packet were not manually implemented.

2.4.3 Bluetooth

While the Raspberry Pi and ESP32 both support Bluetooth 4.2 (Bluetooth Low Energy), it was impractical to use BLE for transmitting one large burst of data as fast as possible because BLE is not designed for this. BLE is designed for periodic updates of specific "characteristics" known by both devices in a BLE connection. Unlike Classic Bluetooth 3.0, BLE does not support socket-like connections and therefore there is no simple approach to connect, transmit limited data, and disconnect using BLE.

For these reasons, the RFCOMM protocol of Bluetooth 3.0 was most appropriate and required very few code changes on the server side from the Wi-Fi experiment since socket programming in C is independent of whether the device is using a Bluetooth RFCOMM socket or a TCP socket over Wi-Fi. However, there were major changes to the code on the client side because, in order to use Bluetooth sockets with the ESP32, the whole state machine must be set up that interprets the headers of all incoming packets from the Bluetooth service and decides what to do with each packet. It was the most complicated program of all the radio schemes used for this experiment.

There was a potentially easier alternative implementation using the Bluetooth UART interface for ESP32 that allows sending packets of data serially using the .print interface, much like printing text to a UART. The reason the Bluetooth interface was done using sockets instead of using the Bluetooth serial print functions is that the serial interface has no windowing at all: the packets are sent and confirmed with ACKs one at a time and therefore do not represent the best possible performance of the radios in this scheme. By sending the packets as soon as the RFCOMM_CAN_SEND_NOW event occurs, it is assured that this is the best possible performance of this particular embedded Bluetooth radio. The TCP requirements, including ACKs and retransmissions, are handled automatically.

2.4.4 Gaussian Frequency Shift Keying (GFSK) RF

The nRF24L01+ is the chip used for the GFSK RF scheme. This is the first radio scheme that is not built into the Raspberry Pi or the ESP32, so the radio had to be attached to the client and server microcontrollers using a 4-wire SPI interface. The client node program is essentially the same for Wi-Fi and GFSK RF except that there is no socket connection. As shown in Figure 2.1, as soon as the ESP32 woke up and initialized the radio, the transaction began with first sending the prepared data's size in bytes that would later be sent over the whole interaction. The radio module used in this work is shown in Figure 2.3.

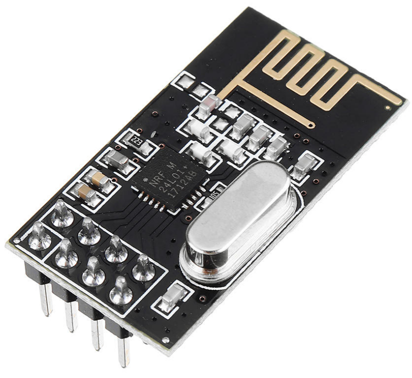


Figure 2.3: nRF24L01+ Radio

2.4.5 LoRa

The RFM69HCW is the radio used for the LoRa scheme. This is also connected to the server and client devices with SPI. The program is the same as the GFSK RF program. For LoRa and GFSK RF, there is a setup time where the radio initializes and is configured for the transmission, but no connect-and-accept exchange as there is with TCP for Wi-Fi and Bluetooth.

2.4.6 ZigBee

The XBee S2C is the radio used for the ZigBee scheme. This is connected via 2-wire UART to the server and client devices. The program is the same as the GFSK RF program except that there is no radio setup. The settings of the radio are stored in non-volatile memory. As soon as the radio is powered on, it is ready to send to any other XBee configured to listen to the same channel the client node will be broadcasting on. As soon as a byte is written to the UART, i.e. the XBee radio, from the ESP32, the packet is immediately transmitted.

All three of the connectionless schemes using non-embedded radios require some time before the transmission actually starts because there are typically some number of retries of transmitting the first payload size packet sent to the server before the radios become completely in sync and acknowledge each other. Therefore there is a non-zero setup time that is comparable to the connection time for two connected schemes. The graphs of the results of the TC experiment show a connection time for all five schemes, with GFSK RF, LoRa, and ZigBee's setup time being used for the reported connection time.

2.5 Transaction Characterization Experiment Results

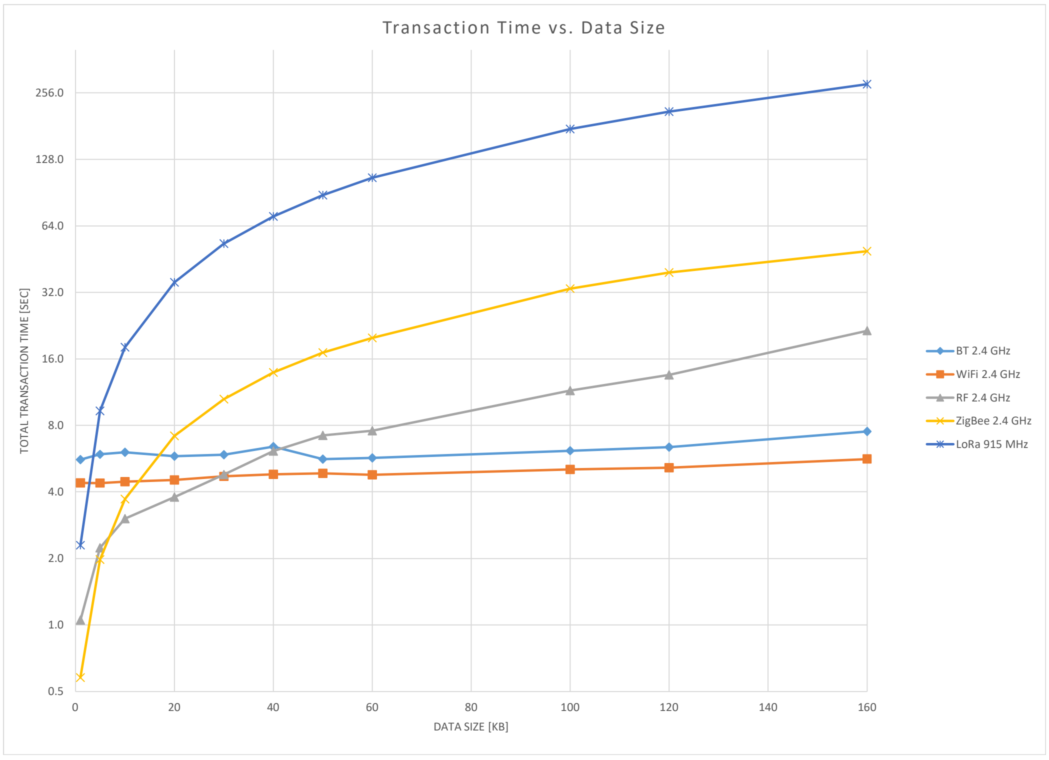


Figure 2.4: Measured Mean Total Transaction Time vs. Data Size. Semilog plot shown to highlight differences in transaction time between radio schemes for small data sizes.

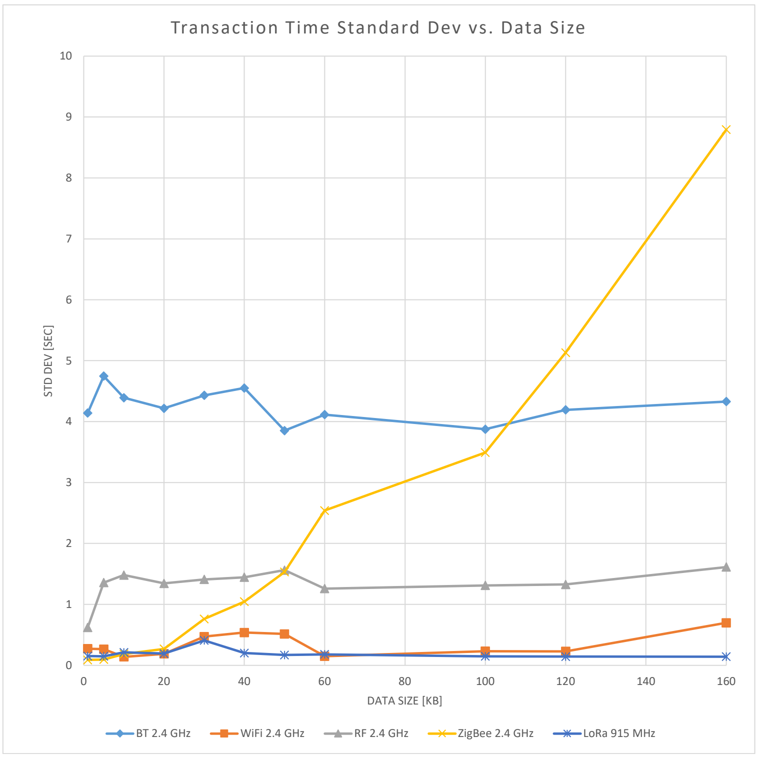


Figure 2.5: Standard Deviation of Measured Total Transaction Time vs. Data Size

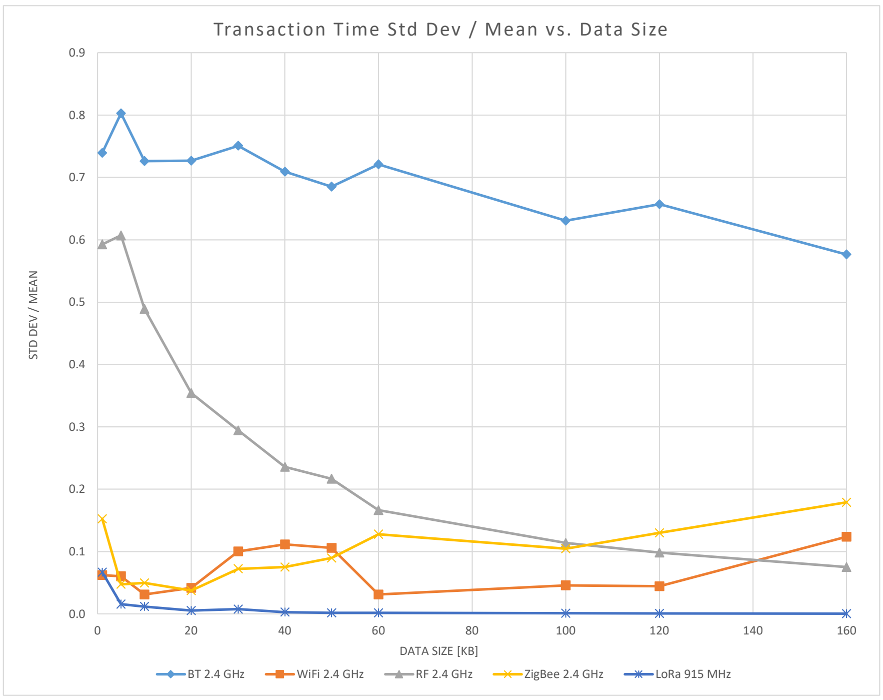


Figure 2.6: Measured Transaction Time - Coefficient of Variation vs. Data Size

	Wi-Fi	**Bluetooth**	**GFSK RF**	**ZigBee**	**LoRa**
Total Transaction Time	4.41 s + 7.1 ms/kB	5.66 s + 8.1 ms/kB	1.28 s + 114 ms/kB	0.968 s + 312 ms/kB	0.596 s + 1.75 s/kB
Transaction Time Std Dev	0.33 s	4.20 s	1.33 s	0.052 ms/kB	0.216 s
Connection Time	4.36 s	5.25 s	1.48 s	0.169 s	0.465 s
Connection Time Std Dev	0.223 s	4.25 s	1.41 s	0.088 s	0.184 s
Transmission Rate	147.1 kB/s	73.5 kB/s	8.81 kB/s	3.2 kB/s	0.572 kB/s
Transmission Time Std Dev	0.40 s	0.50 s	0.41 s	0.052 ms/kB	0.065 s
Approximate Simultaneous Connections	20 – 50	< 10	1	1	1

Table 2.1: Approximate Transaction Statistics from Measurements

Figure 2.4 shows the graph of the mean measured total transaction time for increasing data size. This represents the most important results of this experiment for determining which radio scheme is the best choice for a query vehicle network for minimizing the total transaction time. The mean was taken over 1000 data points for each data size and each radio scheme. The total transaction time is the sum of the connection time and the transmission time. Figure 2.5 shows the standard deviation of the 1000 data points for each test from the experiment. Figure 2.6 shows the coefficient of variation C_V (ratio of standard deviation σ to mean μ, a measure of the spread of a statistical distribution) for the total transaction time. Table 2.1 shows the approximate statistics and derived linear relationship between transaction time and data size that the experiment revealed about the behavior of each radio scheme.

Graphs of the connection time, transmission time, the standard deviation of the those two times, and the coefficient of variation of the two times are shown in Figures 2.7 – 2.12. Representative histograms of the measured total transaction time for 1000 trials per data size for each radio scheme are shown in Figures 2.13 – 2.18. One representative transmission time graph is shown in Figure 2.19.

2.6 Discussion

There are several things to consider when selecting a radio for a particular query vehicle network implementation. The kind of characteristics a radio, and associated transmission protocol, can contribute to the system directly affect the metrics of average transaction time, transaction time variance, and

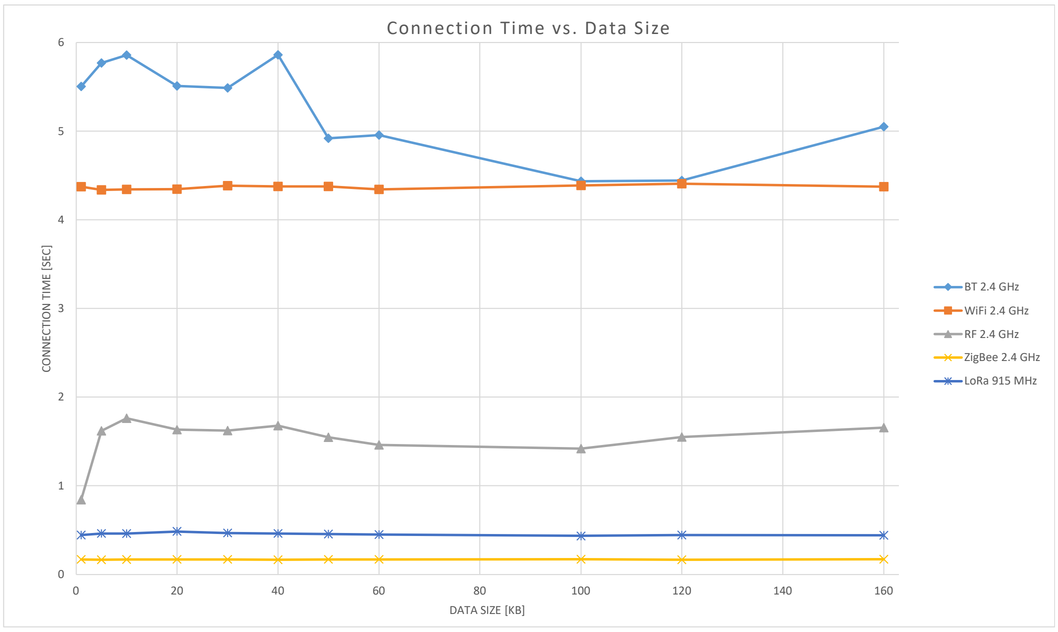


Figure 2.7: Measured Connection Time vs. Data Size

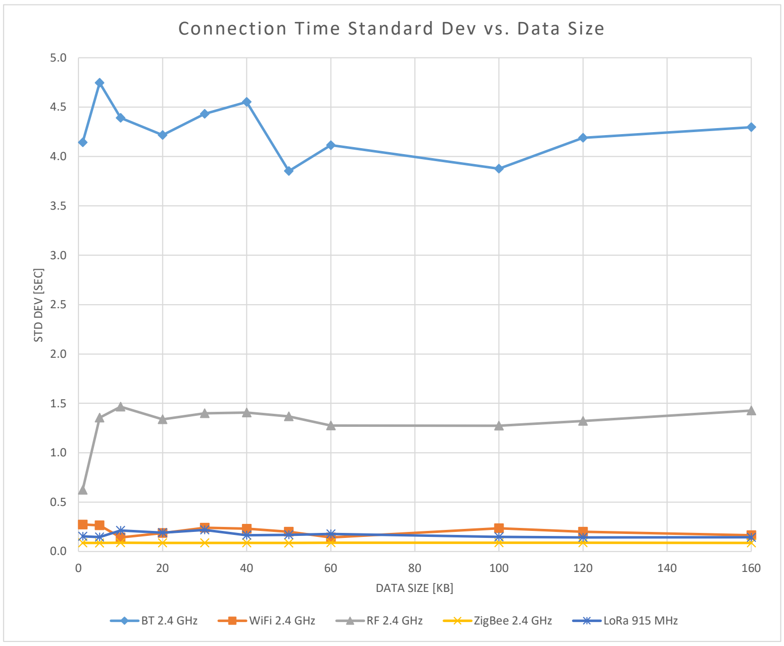


Figure 2.8: Measured Standard Deviation of Connection Time vs. Data Size

implementation reliability.

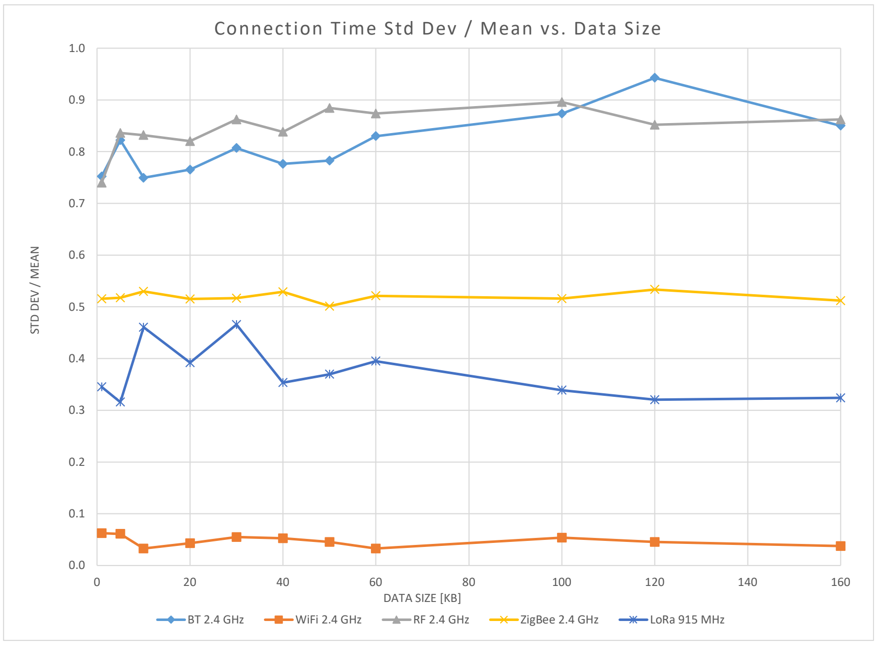


Figure 2.9: Measured Connection Time - Coefficient of Variation vs. Data Size

2.6.1 Radio Characteristics

The first characteristic to consider is whether the radio scheme is connectionless or connected. In theory, the connectionless protocols (GFSK RF, LoRa, ZigBee) should have lower transaction times than the connected ones (Wi-Fi, Bluetooth) because there is very little time between the client node waking and transmitting. However, the data rates during transmission would have to be equal for this theory to hold. Radio initialization time is another factor that contributes to the performance. ZigBee radios have nonvolatile memory for storing the configuration settings so that the radio is ready to go as soon as it powers on. This is why the results in Table 2.1 show ZigBee to have the fastest connection time. For GFSK RF and LoRa, the protocols are connectionless, but the radios still have to be configured every time the ESP32 wakes up because the radio is powered off when the ESP32 is asleep. Wi-Fi and Bluetooth both have setup times on the same order as GFSK RF and LoRa, but the connection time, since they are connected protocols, makes the total time from wake up to the start of the transmission phase four times greater than the setup time of GFSK RF and LoRa. The frequency of all the radios was 2.4 GHz except for the LoRa radio, which operated in the 915 MHz band. Because of this, the LoRa radio scheme has the largest range of all the radios tested in this experiment. A LoRa-enabled server and client node could be 10 km apart and still communicate [31]. The range of each of the other four radio schemes is ~ 30 m.

Simultaneous client connections to the query vehicle are a big factor in choosing the right radio scheme for a particular network as well. The server program forks a new process for each client node that connects to it. For Wi-Fi and Bluetooth, the TCP/IP stack abstracts away the routing of the packets to the intended recipient. For the other three radio schemes, additional code is required to change which client is being listened to or transmitted to. For the three connectionless protocols, it is impractical for the server radio to receive data from more than one client node simultaneously since it requires the server to signal its radio to switch which channel or client it is listening to every time a process in the server program tries to use the shared radio resource. For Bluetooth radios, the theoretical limit is 60 [32] simultaneous connections; in practice, however, the number of simultaneous connections should be limited to less than $\sim 7 - 10$ because Bluetooth is not designed to support downloading files from many connected devices simultaneously. The theoretical limit for simultaneous Wi-Fi connected devices is 255. For Wi-Fi, in practice, it is possible to download from more than twice as many connected devices as Bluetooth simultaneously because the IEEE 802.11 standards were designed to be able to do this. The TC experiment did not characterize the effect of simultaneously connected client nodes, but future work to study this effect should be done. Table 2.1 shows the recommended approximate limits of simultaneously connected client nodes for query vehicle networks.

Transmission rate is the third characteristic that impacts transaction time. The transmission rate of the five radio schemes tested ranges from 0.572 kB/s to 147.1 kB/s. The order of increasing transmission rate is LoRa, ZigBee, GFSK RF, Bluetooth, and Wi-Fi. Bluetooth and Wi-Fi have similar connection times of about 4.5 seconds on average, the standard deviation of Bluetooth's connection time being much higher, and the transmission rate of Wi-Fi being at least twice as fast as Bluetooth.

2.6.2 Metrics

The metrics that the TC experiment evaluated for each radio scheme are connection time, transmission time, total transaction time, and variance in connection time and transmission rate.

Mean transmission time is shown in Figure 2.10 and Table 2.1. The mean transmission time for each radio is predictable and linear, as seen in the representative linear-linear plot of the ZigBee transmission time results in Figure 2.19. The R^2 values for the data in Figure 2.10 ranges from 0.956 (for Wi-Fi) to 0.980 (for GFSK RF), and ≥ 0.993 for the other three radio schemes. The graphs of average transmission time do not

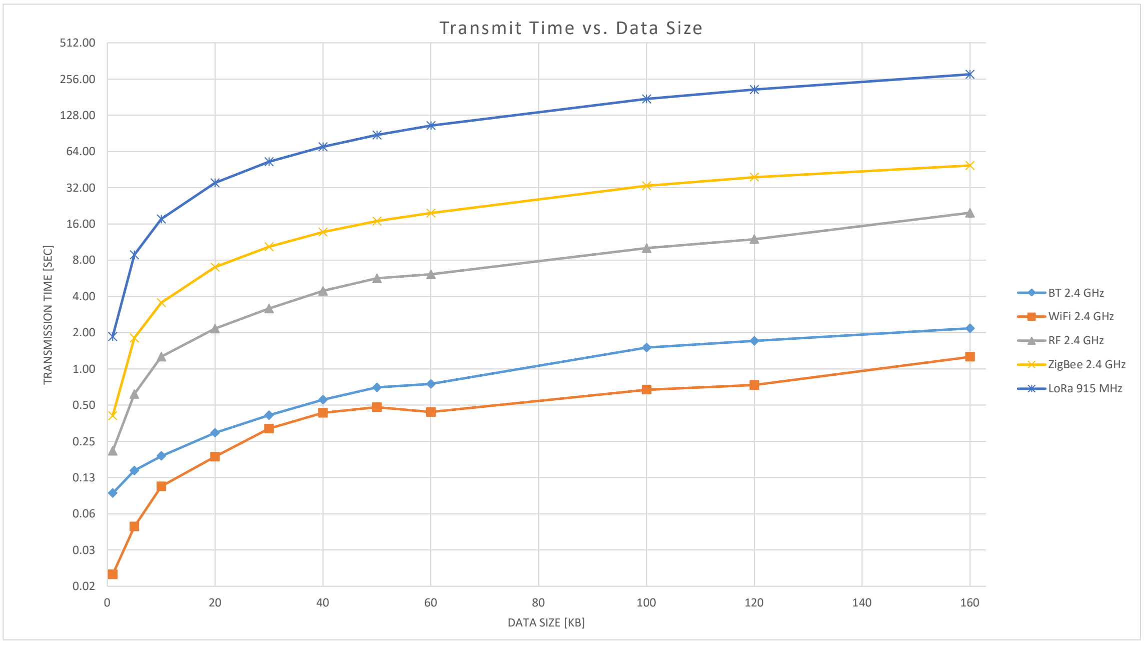


Figure 2.10: Measured Transmission Time vs. Data Size. Semilog plot shown to highlight differences in transaction time between radio schemes for small data sizes.

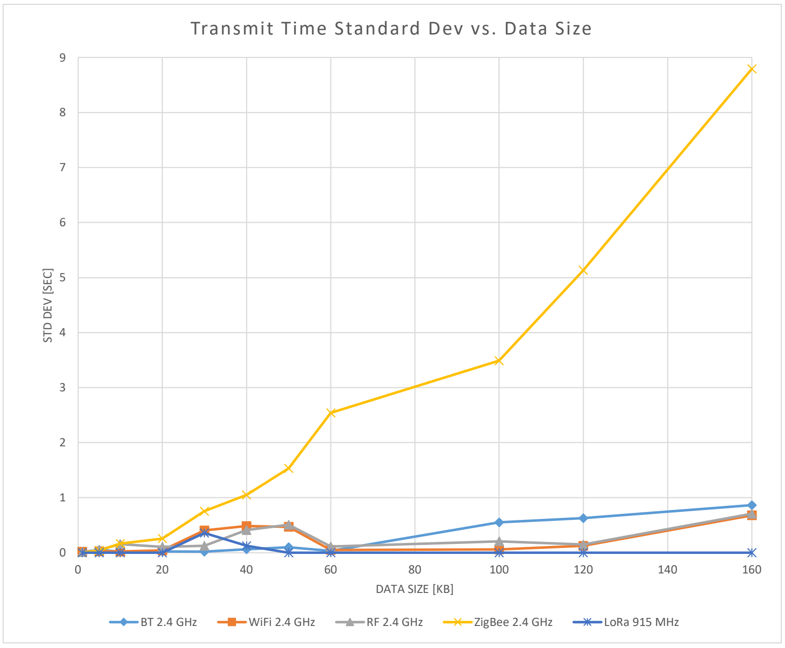


Figure 2.11: Measured Standard Deviation of Transmission Time vs. Data Size

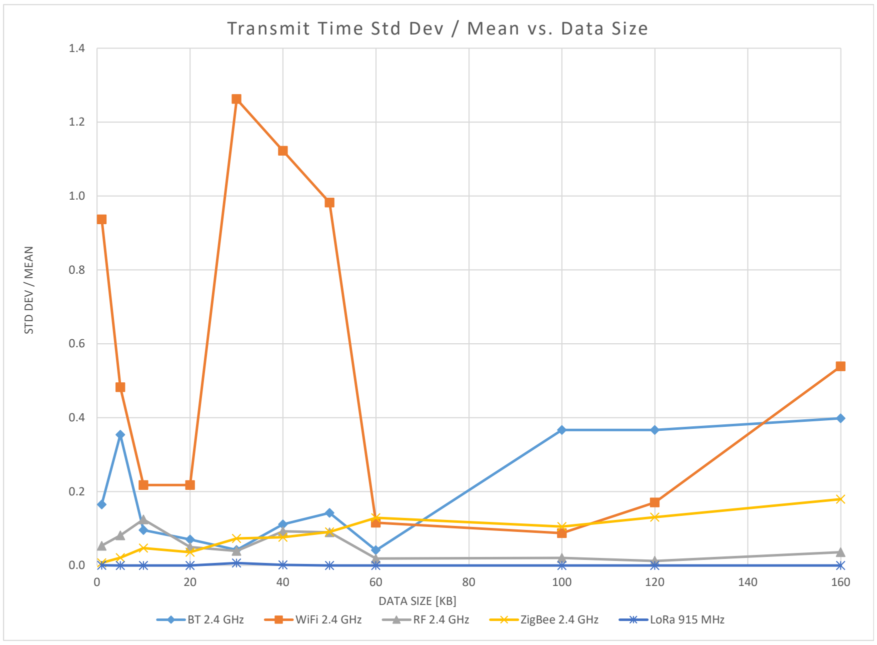


Figure 2.12: Measured Transmission Time - Coefficient of Variation vs. Data Size

intersect each other in Figure 2.10. This indicates that each radio scheme has a distinct and nearly constant transmission rate. ZigBee's transmission time, however, has 2 modes, a fast mode and a slower mode. In the fast mode, the data packets are transmitted and acknowledged about twice as fast between ZigBee radios. The radios will switch from one mode to the other every few minutes as part of the protocol they use to talk to each other and this is not under the control of the ESP32, which is writing to the ZigBee's UART. The effect on standard deviation is discussed below.

Connection time is shown in Figure 2.7 and Table 2.1. This is the time from ESP32 wake up to the start of the transmit phase when the Raspberry Pi server has acknowledged the client's first packet (containing data size). The ZigBee has the shortest setup time because all the configuration settings are stored in nonvolatile memory, unlike the other radios. The Bluetooth and Wi-Fi connection times are the longest because they both use a connected protocol with TCP sockets to communicate.

Mean total transaction time is shown in Figure 2.4 and Table 2.1. Being the sum of connection time and transmission time, it also exhibits a linear trend with a predictable y-intercept at the connection time and a slope the same as the slope of the transmission time graph. Section 2.6.3 discusses this trend and its implications.

The variation in connection time and transmission rate as measured using the standard deviation is an important metric as well. The standard deviation of all the metrics are shown in Figures 2.5, 2.8, and 2.11 and Table 2.1. The standard deviation of connection time shown in Figure 2.8 is independent of data size and roughly constant. The value of the horizontal best-fit trendline for each of those roughly constant graphs is shown in the "Connection Time Std Dev" row of Table 2.1. Bluetooth has a very high standard deviation, 4.25 s, for its connection time, meaning although it takes on average 5.25 seconds to connect to the server, it can often take as long as 9.5 s and sometimes as long as 18 s.

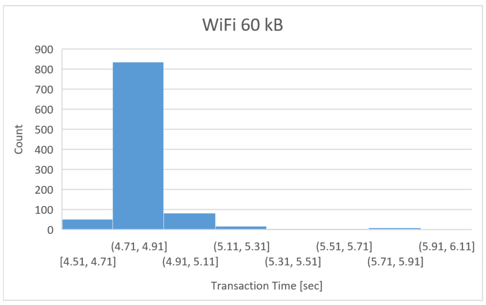


Figure 2.13: Wi-Fi 60 kB measured transaction time histogram of all trials.

ZigBee's standard deviation for total transaction time increases with data size. Because there are two different modes the radio will switch between two transmission rates. It can be seen from Figure 2.11 that the ZigBee transmission time has increasing standard deviation proportional to data size. This is because the longer the total transaction takes, the more likely the radio will switch to the other mode with a different transmission rate in the middle of the transaction. Therefore some of the 1000 trials of each data size use the faster rate, some use the lower rate, and some use both because the radio made the transition during the trial. The standard deviation of transmission rate shouldn't continue to increase indefinitely at higher data sizes however, because if the transaction is several minutes long (longer than the period the ZigBee radio stays in one mode), it is then guaranteed that both modes will occur within the same trial and the variance of the total transmission time should plateau.

Figures 2.9, 2.12, and 2.6 show that for most of the radio schemes evaluated in this study, the distributions of connection time, transmission time, and total transaction time vary somewhat in their spread. Total transaction time (Figure 2.6) is fairly narrowly distributed ($C_V < 0.2$) for all radio schemes except GFSK RF and Bluetooth. C_V for both these radio schemes decreases with increasing data size as the most highly variable portion of the entire transaction (connection time as shown in Figure 2.9) becomes smaller relative to the transmission time. The high variability shown by Wi-Fi transmission time (Figure 2.12) is due to its high data rate making transmission time very small (less than 2 s on average for all data sizes), so variations that are small in an absolute sense will be large relative to the transmission time.

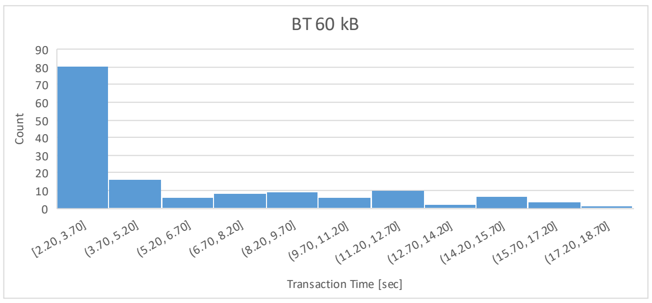


Figure 2.14: Bluetooth 60 kB measured transaction time histogram of all trials.

Reliability is not measured directly by this experiment, but is hinted at by the performance of the radio schemes during the experimental trials. The RF modules were very reliable at syncing and beginning the transmission during the experiment. Large outliers for Wi-Fi, Bluetooth, and GFSK RF experiments, where the modules got into a bad state and never connected at all, were discarded from the shown results. This condition happened rarely with Bluetooth and Wi-Fi, but more often with the RF modules, most likely because the radios were not designed to transmit many kilobytes of data 1000 times in a relatively short period of time and they needed time to cool down or idle between large bursts of usage. When this occurred, the server timed out on the transaction after 1 minute and the client was programmatically restarted to try again. These connection failures were not included as part of the measured transactions.

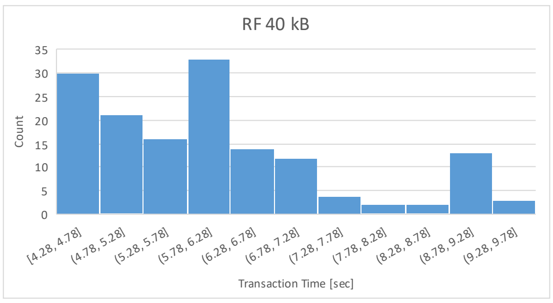


Figure 2.15: GFSK RF 40 kB measured transaction time histogram of all trials.

2.6.3 Analysis

Representative selections of the distributions of measured transaction time from repeated trials of each radio scheme's characterization are shown in Figures 2.13, 2.14, 2.15, 2.16, 2.17, and 2.18. These histograms show all the data points for total transaction time from one of the tested data sizes. Although not all experiments were run to 1000 data points, each radio scheme was tested for at least one data size at 1000 trials. Most of the tests only needed to be run to 160 data points to show a clear trend and for the distribution of the data to converge. For the 1000-point trials, the histogram peaks showed the same distribution as the 160-point trials (but with more data points in each bin proportionally) with the same standard deviation and average.

Figure 2.13 shows that Wi-Fi had a vast majority of the transaction time results falling in one bin of width 0.2 seconds with only a few trials (<200) falling outside that bin. Figure 2.16 shows that LoRa is even more predictable with less than 10 trials falling outside a bin of width 0.5 seconds, which is quite tightly packed considering that LoRa's transaction times were much greater than all other tested radio schemes due to its low transmission rate. Figure 2.14 shows that Bluetooth's results are much less predictable, with a majority of the transactions falling on one bin of width 1.4 seconds but many falling outside that range in a long tail. This illustrates the much larger standard deviation of the Bluetooth results caused by the highly varying connection time. Figure 2.15 shows that GFSK RF has very little predictability about how long the transaction will take within the bounds of possible times. The specific results in Figure 2.15 show that for 40 kB, the GFSK RF transaction will take between 4.28 and 9.78 seconds, but no discernable pattern as to which time in that range is most likely to occur.

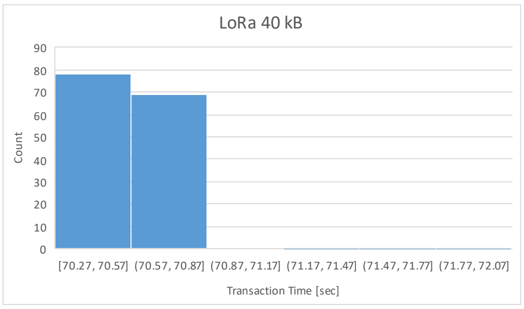


Figure 2.16: LoRa 40 kB measured transaction time histogram of all trials.

Two sets of results are shown for ZigBee in Figures 2.17 and 2.18 to illustrate that the standard deviation increases proportionally with data size. In both experiments, there are two large peaks in the resulting histogram, with many data points in bins of shorter time. This is again caused by the two modes of ZigBee's transmission phase. Very few trials had a high proportion of the transmission phase using the fast transmission rate, and most of the trials had a large proportion of the transmission phase that used the slower transmission rate. The protocol that the XBee S2C radios use switches between the fast and slow transmission rate without the host microcontroller being able to control the transmission rate or when to switch between the modes. Figure 2.19 shows more clearly than the semilog plot of transmission time shown in Figure 2.10 that the transmission phase duration vs. data size is linear, not just for ZigBee, but for all tested radio schemes.

The most important results of this experiment for selecting a radio scheme for minimizing the total transaction time in a query vehicle network are shown in Figure 2.4. Wi-Fi is the radio scheme that yields the shortest transaction time for client nodes that need to transmit more than 30 kB each time the query vehicle passes by. GFSK RF gives the shortest transaction time for >5 kB and ≤ 30 kB. ZigBee is faster than GFSK RF for 5 kB or less. However, unless a query vehicle is guaranteed to only need to receive 5 kB or less for almost all transactions for the lifetime of the network, GFSK RF can be a better choice considering that it is the fastest radio scheme for 30 kB or less and only half a second slower for the 5 kB range in which ZigBee is faster.

Most retired mobile devices will only have Wi-Fi and Bluetooth in addition to their cellular radios, so the results of this work indicate that Wi-Fi would be the better choice for query vehicle data harvesting from distributed sensors made by reusing mobile devices. This is largely due to the higher connection time and and higher connection time variance. While the absolute difference in mean total transaction time may not impact significantly the data upload latency for a ground-based query vehicle, the difference is likely to be significant for a much faster UAV query vehicle. The long tail of the transaction time distribution for Bluetooth (Figure 2.14) suggests that the worst-case upload latency for Bluetooth will be approximately $3\times$ worst than for Wi-Fi, which may be unacceptable for some applications.

2.7 Related Work

A majority of papers researching data mule topics focus on 1) applications for data mules, 2) network topology in systems that contain query vehicles, nodes, and servers, 3) scheduling the deployment of the vehicles or mobile nodes, and 4) path planning for most efficiently traversing the array of nodes using one or

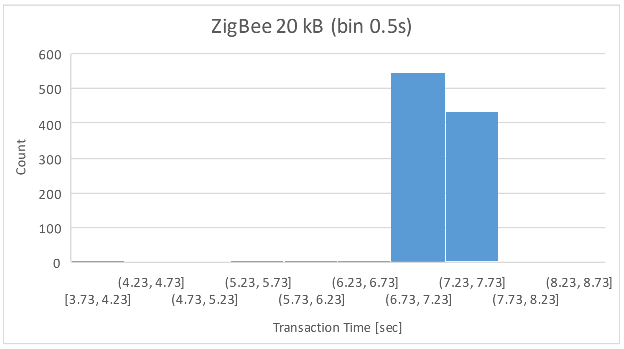


Figure 2.17: ZigBee 20 kB measured transaction time histogram of all trials.

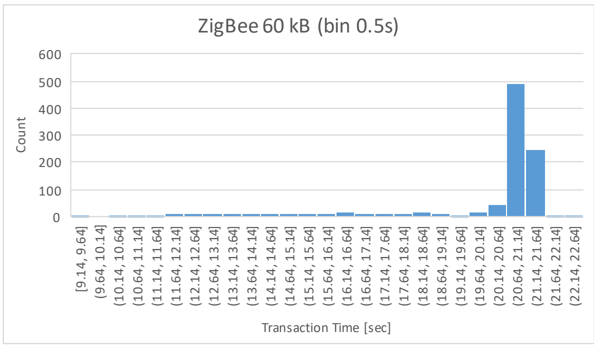


Figure 2.18: ZigBee 60 kB measured transaction time histogram of all trials.

more query vehicles. Much of this work is orthogonal to the wireless network transaction characterization presented here.

2.7.1 Data Mule Applications

Among the many data mule relate papers, a popular topic of discussion is the usefulness of such networks and specific applications where a query vehicle setup might be ideal. A proof of concept for a system of query vehicles that demonstrates the reduction in energy consumption at the sensor nodes in a field of such nodes is presented by Tekdas et al. [33]. Another experiment, presented by Palma et al. [34], provides a set of measurements for transferring data between sensor nodes on a ship and an aerial drone. This experiment used Wi-Fi and four different file transfer protocols to show that the file transfer protocol does matter, the DTN2 protocol being the best for their setup. Even more important than the protocol, the trajectory of the drone had the greatest impact on the success of the file transfers. In their paper about hiding the location and securing the data of wireless sensor nodes, Raj et al. [35] simulate and analyze a theoretical data mule system that they show can make the sensor nodes' locations anonymous at the cost of adding latency to the collection of the data. Coutinho et al. [36] present a system of boats in the Amazon region of Brazil that have regular routes throughout the area and carry data between a central server at the state capital and various nodes along its route. These locations along the route don't have internet access and rely on the boats for transmitting important medical data. To address link or communication failures in a normally-connected large-area network of sensor nodes, Crowcroft et al. [37] present a data mule solution for visiting the neighborhoods of such failed network nodes and recovering or delivering the data with mobile nodes until the network has been repaired. Their paper is a mathematical analysis of a simulated solution to such problems. All of these works demonstrate the promise of query vehiclenetworks to complement traditional wireless networks and therefore motivate strongly the present work.

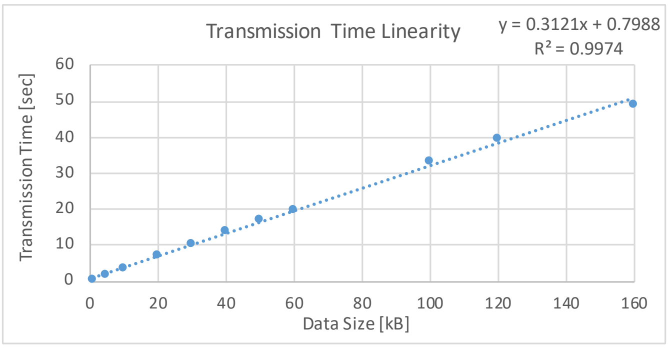


Figure 2.19: ZigBee Mean Measured Transmission Time vs. Data Size

2.7.2 Data Mule Scheduling

Another very common paper topic for data mule research is scheduling the deployment of the query vehicles or mobile nodes, the collection of data from the static nodes, and the delivery of that data to the central server that needs it [38, 39, 40, 41]. The transaction times studied in the present work can be incorporated into the scheduling of query vehicles to further optimize the overall performance of the system.

2.7.3 Data Mule Path Planning

More than any other topic, the research that is most often published about data mule networks explores path planning. It is a very complex problem to plan the optimal path for one or more query vehicles to travel in order to most efficiently traverse all nodes in the array and deliver the data to a central server. All of this must also happen while avoiding collisions with other query vehicles or obstacles while not redundantly visiting any nodes more than once [42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55]. The study presented here is orthogonal to the line of research on path planning.

2.7.4 Wake Up Signals

Several papers study and present solutions to waking sensor nodes that spend most of their time in very low power states of processor sleep, from a wireless signal that a query vehicle might broadcast to the client nodes. There are many protocols proposed for how the wake-up transmission should work, many designs of the circuit responsible for listening for the signal and waking the node, a general agreement that the radio should listen with a 10 – 50% duty cycle and not constantly, and many analyses of the low power theoretically achieved by such systems [56, 57, 58, 59, 60]. Wake-up time would directly add to the total transaction times characterized in this study, but the related work suggests the additional latency would not be significant. The additional waking time if using the protocols determined in those works is 0.5 – 5 ms. Assuming the worst case of 5 ms, this time is between 0.002% and 0.1% of the total transaction time in 98.6% of the tested combinations of radio schemes and data sizes, and under 0.8% total time in all cases.

2.7.5 Wireless Protocol Characterization

There are many works on characterizing wireless protocols. Most of those characterize new radios or the physical channel itself, many analyze custom selective retransmit or error correcting protocols, and many characterize specific attributes or environments for common protocols like TCP over Wi-Fi. Corvaja et al. [61] measured quality of service for Wi-Fi/Bluetooth overlay networks where the devices in a star network are using Bluetooth to communicate small distances to peripherals and Wi-Fi to connect to the bigger nodes higher up in the star hierarchy. Several papers present and compare the specifications and theoretical performance of several popular wireless protocols with no experimental data [62], but with models and graphs of the behavior in typical use. One in particular presents a graph of the theoretical transaction time for ZigBee, Bluetooth, and Wi-Fi [63]. The paper predicts that Wi-Fi is the fastest for overall transaction time, Bluetooth is 50 times slower, and ZigBee is 3 times slower than Bluetooth. The TC experiment's results are in agreement with those rankings, but show that low power Wi-Fi modules, like what would be on a sensor node in a query vehicle network, are not 50 times faster than Bluetooth 3.0, but only about 2 times faster, with ZigBee being 24 times slower than Bluetooth. Duarte et al. characterized a specific narrowband full-duplex protocol that they implemented. Their analysis focuses on the protocol's ability to compensate for noise and errors caused by the physical channel [64]. None of the previous works focus on what the present TC experiment attempts to characterize: the differences in total transaction time, transmission time, and connect/setup time under real experimental conditions with the radios in each test setup with the same environment and layout for fair comparison.

2.7.6 Data Mule-Specific Wireless Networking

Jain et al. [65] discussed the mathematical relationship between these parameters: properties of the sensor nodes, query vehicles, and radios used for communications between sensor node, query vehicle, and server (or "access point"). The metrics that these parameters affect are power, effectiveness of data delivery, and latency from node to server. The mathematical analysis and proofs in their paper show the energy savings that using a query vehicle can achieve and the stability conditions for a network of nodes and "MULEs" operating at known data generation rate and collection frequency. This kind of stable data mule network is dependent on the interaction between nodes and the MULEs being as brief as possible in order to enable the MULEs to pass by without stopping and simultaneously collect data, which is why finding an ideal radio scheme, as studied in the present work, for such a functioning network is so important.

Anastasi et el. [66] modeled a flyby scenario similar to the one described in the present work and proposed a protocol, ADT, that attempts to minimize energy consumption of the client node by minimizing the transmission time. ADT predicts when the query vehicle will be the optimal distance from the client node to start transmission such that the fewest packets are corrupted or lost. Their analysis shows that a "naive protocol" that transmits as soon as it is woken by the query vehicle, as assumed in the TC experiment presented here, is not as energy efficient as a protocol like ADT that waits until the transmission distance is more ideal, but "may be more robust" because of having more time to transmit to the passing vehicle. ADT involves the sensor node predicting the optimal time to transmit the data based on how big the data is, and data it recorded from past interactions with the MULE about "contact time" (how long it was in communications range during those interactions). The transmission happens in the middle of the contact time range to minimize the average distance between MULE and sensor node during transmission. Although their paper [66] is a purely theoretical analysis, the authors use data from another paper of theirs in their mathematical model of ADT's performance [67]. This experimental data is taken on real sensor nodes in an urban environment with a MULE collecting data at two different velocities. The paper shows that packet loss is roughly proportional to distance as the MULE is passing by the sensor node [67]. The TC experiment in this paper does feature the client node transmitting as soon as the radio and processor are woken and ready to transmit. The TC experiment did not attempt to find an ideal strategy for when to transmit after the node wakes, but rather what kind of protocol and radio scheme should be used to minimize the total transaction time. This would affect the "steady phase" of ADT if the radio scheme chosen from the results of the TC experiment were used with ADT.

Anastasi et el. [68] also wrote about the "data transfer phase", which corresponds to the transaction being characterized in the TC experiment. They simulated the transactions, taking into account the packet loss expected from a passing query vehicle. Their paper presents a mathematical model of the "data transfer phase" with parameters for "Contact time", "Waiting time", and "Discovery time", showing how they affect the energy consumption per byte for transactions in such a network. Their simulated experiment assumes a Chipcon CC1000 radio with bit rate of 2.4 kB/s during the transmit phase. Anastasi et al. show in their paper that a window size ≥ 5 packets (12 packets for the radio they chose) is ideal for reducing the total transaction time. They also show that having the sensor nodes only listening for the MULE 10% of the time (10% duty cycle) reduces energy consumption by a factor of 11, but is not improved by further reducing this duty cycle. The windowing is very important in a radio scheme. Two of the radio schemes (Bluetooth and Wi-Fi) characterized by the TC experiment presented here use TCP, which contains a windowing protocol. The total transaction time of the other three schemes investigated would be improved slightly by adding a windowing protocol to the transmission phase of the transaction, but ultimately the slower transmission rate will always prevent those three schemes from having a shorter transaction time than Bluetooth and Wi-Fi for larger packet sizes.

2.8 Conclusion

This experiment has shown for at least the tested radio schemes that Wi-Fi is the fastest option for transferring >30 kB of data in a single pass of the query vehicle, GFSK RF is most suitable for >5 kB and $\leq$ 30 kB payloads, and ZigBee for 5 kB or less. For query vehicle networks with a similar topology to that simulated by the TC experiment, Wi-Fi is the best choice of radio scheme for many reasons, with the first being speed. The transmission rate is faster than the other radio schemes and the total transaction time is shorter for more than 30 kB of data. Second is cost, with Wi-Fi radios being available for $4 or less and many microcontroller modules like the ESP32 having Wi-Fi built-in. Effectively all retired mobile devices that can be reused for distributed sensing incorporate Wi-Fi. Third is multiple simultaneous connections, since Wi-Fi is specifically designed to be able to stream data from multiple connected devices, much more so than other radio schemes studied here.

Even though LoRa is by far the slowest radio scheme in terms of total transaction time, because it has a theoretical range of approximately 10 km, it is possible to construct a query vehicle network with LoRa radios where the query vehicle doesn't have to pass nearby the client nodes. The vehicle (ground or aerial) could park near the center of the array and collect data from all nodes without moving. If the area of the whole array of sensor nodes is small enough, there doesn't need to be a query vehicle at all; the central server connected to a LoRa radio could query the client nodes whenever needed. However, this construction is very likely to run into problems with waking the nodes because the technology of waking sensor nodes wirelessly, discussed in Section 2.7, does not work from a distance of one kilometer.

2.9 Future Work

The present work established that there is significant variation in transaction time for communicating data from a single sensor node to a query vehicle in a controlled environment. The next steps would be to (1) get more data about the proposed radio schemes for use in a query vehicle sensor network in a deployment such as a network of multiple sensors in a field of crops or a warehouse containing sensors, (2) reading from multiple sensor nodes simultaneously, and (3) adding a windowing protocol to each of the three connectionless radio schemes. A field experiment would be a good environment in which to test interference from reflections and obstacles between server and client radios. Doing both a field of crops and a warehouse would give insight about which kind of radio scheme would be best for specific environments. Having the query vehicle read from multiple sensor nodes at the same time would be required in a full implementation, and would reveal even more about how each radio scheme could be used. It is likely that Wi-Fi would win out over the other proposed radio schemes in a test of simultaneous clients, followed by Bluetooth, because Wi-Fi and Bluetooth radios are designed to have simultaneous independent connections in a star network topology with the server at the center. It is quite simple to fork a new process within the C++ server program for each socket connection for these two radios and have the server deal independently with each one in parallel processes (see Section 3.9 for an application of this proposed server software architecture to a distributed computing system). It is nontrivial to do so with the other three radios studied in this work. A windowing protocol allows multiple packets to be sent one after the other without stopping to wait for the acknowlegments (ACKs) until a certain number of packets (the window size) have been sent with no ACK response. Windowing is one of the reasons Wi-Fi and Bluetooth had better transaction time performance compared to the others, because windowing is a standard part of TCP and the selective reject protocol.

Chapter 3

Cluster: A Distributed Computing System Using Retired Mobile Devices

This chapter explains the Cluster project design and operation in detail. Cluster development involves designing, implementing, and characterizing a distributed computing system composed of a heterogeneous collection of retired smartphones and tablets. These are devices that are used, at the typical end of their consumer lifespan (retired), but are still functional, and would otherwise be recycled for their raw materials.

3.1 Motivation

The first major reason to consider building a distributed computer out of mobile devices is the environmental impact. Repurposing used devices avoids the carbon footprint of manufacturing new ones and reduces e-waste, as discussed in Section 1.1. Section 5.5.1 presents quantitative estimates of Cluster's carbon footprint based on the Computational Carbon Intensity metric defined by Switzer et al. [26].

The second motivation for this project is that retired smartphones and tablets are very cheap and often very compute-capable, low-power computers. This motivation is similar to the insight that motivated the "pile-of-PCs" architecture for Beowulf scientific computing clusters: mass production of consumer-grade

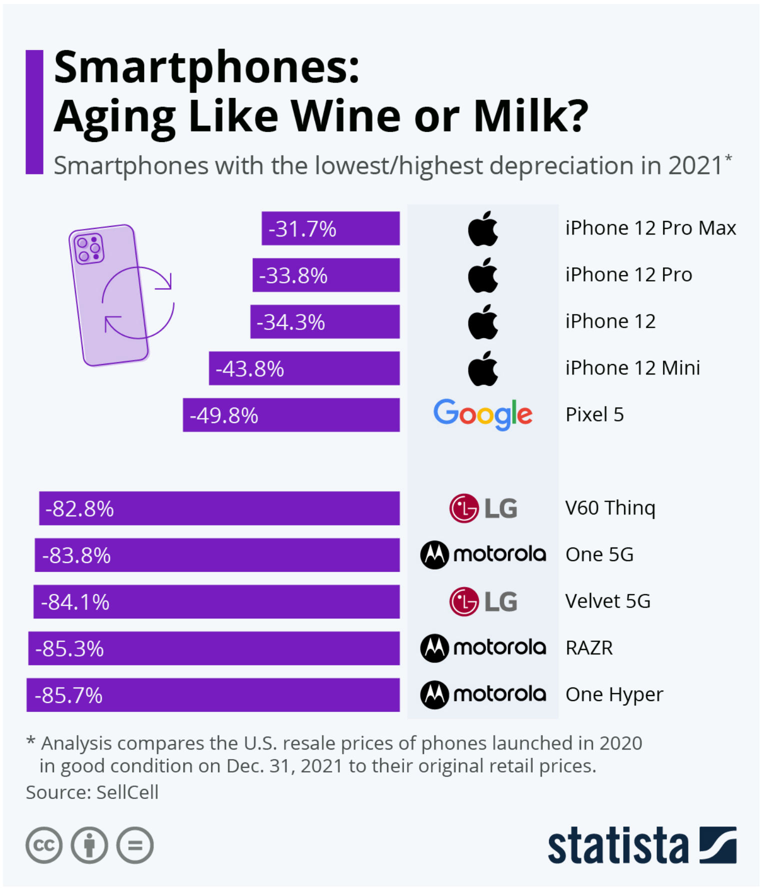


Figure 3.1: Smartphone one-year depreciation rates for models launched in 2020 [4].

personal computers and integrated circuit performance scaling through Moore's Law had made PC-based workstation-LAN distributed computing clusters competitive from a price-performance standpoint with conventional supercomputers [69]. Similar efforts have utilized Raspberry Pi boards [70]. While Raspberry Pi and PC-based Beowulf clusters tend to be homogeneous in both hardware and software and are constructed from new computers, Cluster takes the price-performance insight a step further by using a heterogeneous collection of retired mobile devices. Performance and power consumption are important considerations also. Smartphones and tablets cost between \$500 and \$1200 when new because they are powerful computers with modern CPUs and GPUs. They are designed specifically to be low-power computers capable of running a wide variety of games and programs (apps) while being powered by a battery for a whole day. However, smartphones depreciate quickly, as shown in the infographic in Figure 3.1, with annual depreciation rates between 32% and 86%. Devices that are at least 5 years old are of little perceived monetary value, typically under $50 [71, 72], and as such people are often willing to donate their devices since they have no other use at the end of their typical lifetime of 2 – 4 years. Many of the devices utilized in Cluster-related research were donated, see Sections 3.11 and A.2.

Privacy is also a major concern for many users of distributed computing applications. There are many entities including companies, non-profit organizations, and communities with common interests that, if it were affordable, would buy their own compute resources to have their data completely contained within either an offline system or a private network, one that is not dependent on the large commercial cloud services companies. A cloudlet made from retired mobile devices would be a low power distributed computer with minimal cooling requirements. Since there is no extra hardware required for cooling and each compute node consumes only 65 – 200 square centimeters of area depending on how much space between nodes is chosen for the design, the system would also be a very small-area implementation.

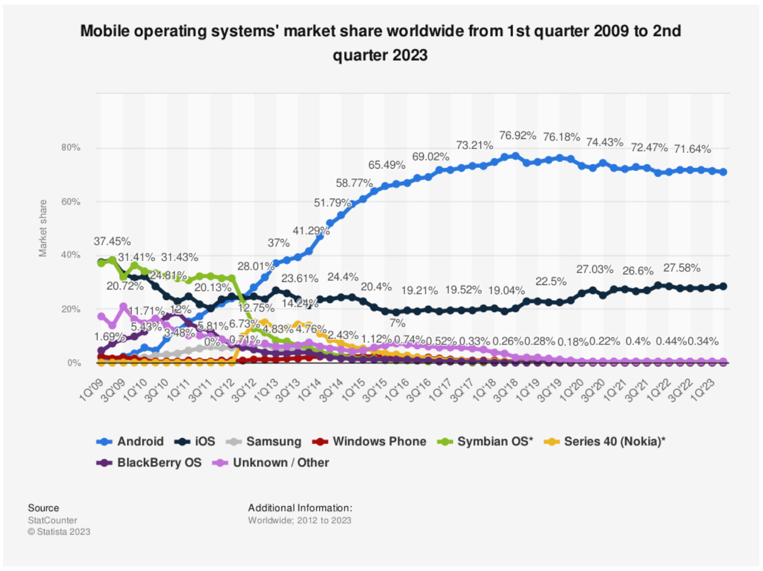


Figure 3.2: Worldwide market share of mobile operating systems [5].

Because there are so many different types and brands of mobile devices, initially there were concerns about software compatibility. Several other works have investigated installing a Linux operating system on a limited number of Android device models that are able to run it. This approach does not work for iOS devices at all since Apple does not allow direct access to the hardware or reprogramming of the operating system. Excluding iOS devices from a reuse strategy substantially reduces the number of devices that can be reused, limiting the impact on reducing e-waste and carbon footprint. The Cluster approach uses an app installed on the devices and therefore any Android or iOS device newer than 2012 (more details in Section 3.4) is compatible. This approach therefore accounts for 99% of all smartphones in 2023 as shown in Figure 3.2 [5].

3.2 Limitations

In addition to the reasons above advocating for a system like Cluster over traditional distributed computer architectures, there are also limitations to such a design. The limitations of Cluster are those of Wi-Fi power, bandwidth, scalability, reliability of the devices, and the effort of rewriting any program for the Cluster platform. First, smartphones and tablets do not have wired networking capability, they typically have only Wi-Fi or cellular data and wireless transmission uses more power than wired transmission of the same data [73, 74]. Using cellular data would introduce the possibly of the devices not needing to be on the same LAN, but also introduces more network delay and more power to transmit and receive packets over the network. It also would require a redesign of the socket interface for the client software to be able to locate the server node and connect to it. The proposed implementation of Cluster utilizes Wi-Fi only. Shahrad and Wentzlaff [25] proposed using a wired network of USB cables and hubs for their smartphone-based server and this option for Cluster could be explored as future work.

Second, by the very nature of these retired devices being at the end of their typical user lifetimes, the hardware will never be as reliable as new mobile devices or computers. The client nodes of Cluster would have to be replaced more often, but this is somewhat compensated by the relatively low or zero price of the used devices.

Third, unlike a network of Raspberry Pi devices or server-grade computers, mobile devices which are running the Cluster app are limited, in the current implementation, by what the app is programmed to do. This means Cluster cannot be used to run an existing parallel computing program, all programs that it runs must be ported to Python, Swift, and Kotlin / Java in order for Cluster to execute it. While this does not restrict what kinds of programs the cluster can run, it does introduce significant effort for rewriting any existing program to function on the cluster. The focus of the present work is on tasks that would support cyber-physical systems, so the restriction to only tasks specifically programmed for Cluster is appropriate to this application domain.

3.3 Research Questions

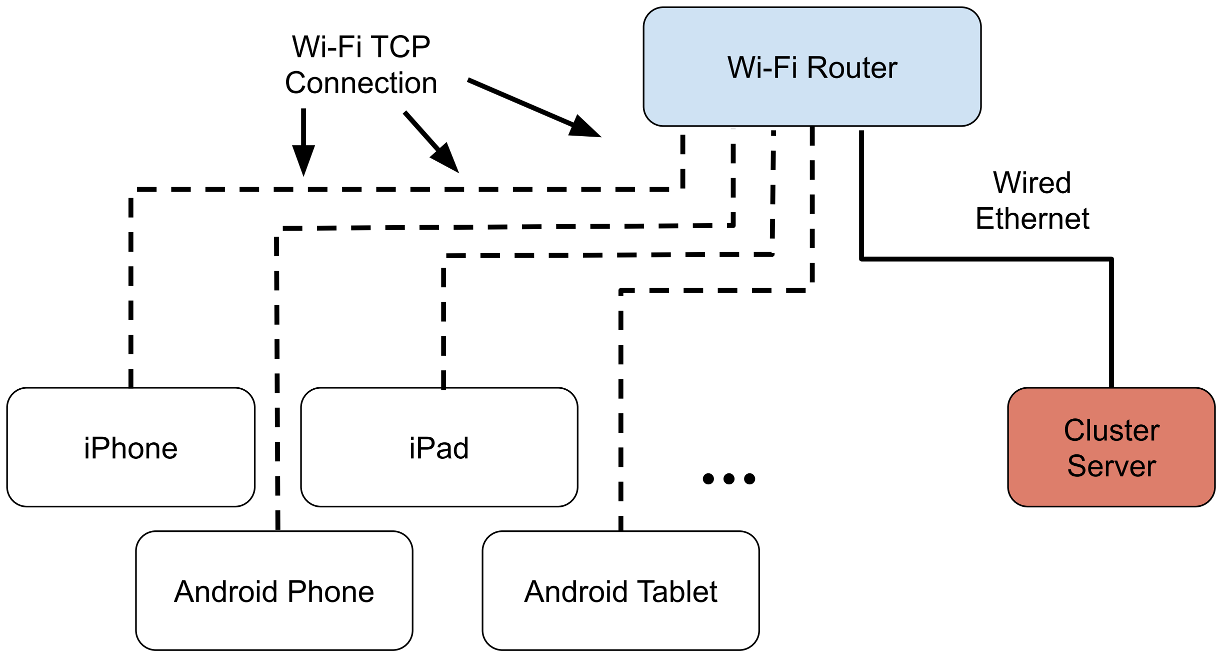


Figure 3.3: Cluster Network Block Diagram

Cluster is a local area network (LAN) distributed computer made of retired mobile devices, which use Wi-Fi to connect to a wireless LAN router and a server computer wired to the router. This is shown in Figure 3.3. The research questions this work is meant to investigate are the following:

1. Where is the computing bottleneck in Cluster: the server program, the compute capabilities of the client nodes, the Wi-Fi router, or the Wi-Fi module on the server computer?

   A series of experiments and synthetic compute tasks were designed and implemented to investigate how the cluster performs under the constraint of these various performance bottlenecks.

2. Does using multiple processes instead of a single multi-threaded process on the server program improve the program's scalability and how does it affect overall performance?

   Two versions of the server program were designed and implemented to test the hypothesis that compute tasks would finish more quickly on Cluster if the program occupied multiple cores of the server computer's processor instead of just multiple threads on a single core.

3. What is the limit of how many devices can operate in Cluster until performance no longer improves?

And does this limit depend on what kind of task is being performed?

To answer this question, many experiments were run to test the performance of each of the four compute tasks implemented for Cluster and the execution time and network usage data were tracked for each run and for an increasing number of connected mobile devices. The experiments were run at least 20 times for each quantity of connected devices.

4. How is the performance of Cluster affected by changing the wireless environment from an open space to a shielded space (i.e., enclosing Cluster in a metal Faraday cage)?

   Two sets of experiments were run in a large industrial metal Faraday cage in the basement of a university research building. These experiments were set up to have identical conditions to all other characterization experiments with the only change being the room they were run in.

In addition to these questions about Cluster itself is the question of how Cluster compares to other distributed computers of similar architectures. Thus, this work is compared to related work in Section 5.8.

3.4 Software Design

Distributed computing systems typically involve many computers housed in the same room on the same wired LAN. A common way to write a program that runs on many machines concurrently on the same LAN is using the Message Passing Interface (MPI) [75]. The MPI library is available in many common programming languages like C++ and Python. However, it is not available for iOS or Android apps as of September 2023. It does make sense that it is not implemented for mobile platforms since the proposed Cluster is one of only a few LAN-based distributed computer implementations for mobile devices and other implementations have replaced the native mobile OSes with Linux. Any distributed computer that uses retired mobile devices would likely dedicate all the devices to the computation task in order to maintain the cluster's connectivity and guarantee processor availability during the task.

Since using MPI was not an option without implementing the protocol from scratch for mobile OS platforms, a custom protocol was implemented to suit the applications targeted by Cluster. A custom packet header, shown in Figure 3.5, was designed alongside the program flow for server-client interaction that governs all the mobile devices. The packet header's design is discussed in detail in Section 3.6.

The proposed device cluster consists of a network of iOS and Android smartphones and tablets, each with the Cluster app installed, governed by the Cluster Server program installed on a single sufficiently powerful computer to handle managing many threads or processes for all the connected devices. The goal was to be able to demonstrate several distributed computing tasks running on the cluster and compare their performance to a common inexpensive cluster computing implementation, i.e. a wired network of Raspberry Pi computers. The computing experiments for this research are described in Chapter 4 and inspired primarily by tasks associated with cyber-physical systems applications.

The Cluster app is written in Swift 5 (https://swift.org/) for iOS and Kotlin (https://kotlinlang.org/) for Android. The multi-threaded server program is written in C# for Windows only (https://learn.microsoft.com/en-us/dotnet/csharp/). C# is a popular language for writing multi-threaded programs with a graphical user interface (GUI). The multi-process cross-platform server program is written in Python (https://www.python.org/). Performance of the Python version of the Cluster server, where a new process is spawned for each device connecting to the server, is discussed in Chapter 4 and the operation is discussed in Section 3.9.

There are two main types of parallel computing tasks evaluated on Cluster, independent and dependent. Independent tasks do not require any collaboration between the cluster node devices. The computing nodes are given their part of the task and can work on it until they are finished without requiring any data from any other nodes to finish, only communication with the coordinator (or server) to receive more work and to return the finished work. Dependent tasks require passing data between the nodes themselves. In this case, the inputs to the programs running on a node are at least somewhat dependent on work being done by other nodes.

3.5 Performance Metrics

The metrics for measuring the performance of the Cluster network of devices are those that can be readily compared to an equivalent cluster of Raspberry Pi or other devices achieving the same task.

The metrics are:

- **Time to completion**: elapsed time to complete a specific task

- **Mean time to failure (MTTF) / mean time between failures (MTBF)**: of a device in the network, usually the app crashing or the TCP connection becoming inoperable
- **Cost**: of the devices involved, including server and networking hardware
- **Energy Consumption**: of all devices and the server node during completion of a task and idle mode (resting)
- **Speedup vs. number of devices**: relationship between speedup of a task and number of active computing nodes, may continue to increase in some circumstances for an increasing number of devices
- **Task throughput**: quantity of work (e.g. number of images processed) completed per unit time for specific tasks
- **Scalability**: how many nodes can the system be scaled to before performance saturates? How close can the proposed architecture approach the level of performance of other architectures?
- **Computational Carbon Intensity** (CCI): the CO_2-equivalent released per unit of computation work

3.6 Cluster Packet Header Design

When writing code for passing data across a network between two computers, the protocol of choice is often the Transmission Control Protocol (TCP) for simple and easy interactions. The standard TCP libraries are well-understood and well-used for a variety of applications and are supported on general-purpose and moble OSes, but there are some disadvantages as to how the data is received and organized in the queue of packets that come in on a TCP socket on any network-enabled device. The payload is the data that needs to get from one device to the other and the headers are extra bits that are added onto the data packets that allow the packet to be organized and labeled and transported accurately to its destination. When a packet is recieved on a TCP socket, all the headers are removed by the network hardware before any program sees the packet. The abstraction that happens with TCP libraries means that only the payloads are stored in the TCP socket queue, without any boundaries between one TCP packet's payload and the next. Because there is no indication from an application level of how big the packets were when they arrived, what kind of data the packet contains, or what the purpose of the packet is, extra data needs to be included in the payload of all packets sent to and from the devices to keep track of this information. The Cluster packet header was designed for this purpose and its structure is shown in Figure 3.5.

The structure of the packets that will be sent across the network is crucial to the operation of the network. Packets in a computer network are organized by the OSI stack, a standard of metadata that gets added onto the payload before it can be sent out on the network. The metadata is organized into a hierarchy of headers added at each level of the stack as the packet leaves the application on the devices and is sent to various networking hardware. The network switches and routers are responsible for getting the packet to its destination, tracking its progress and origin, and verifying its validity since the data can be corrupted in transit.

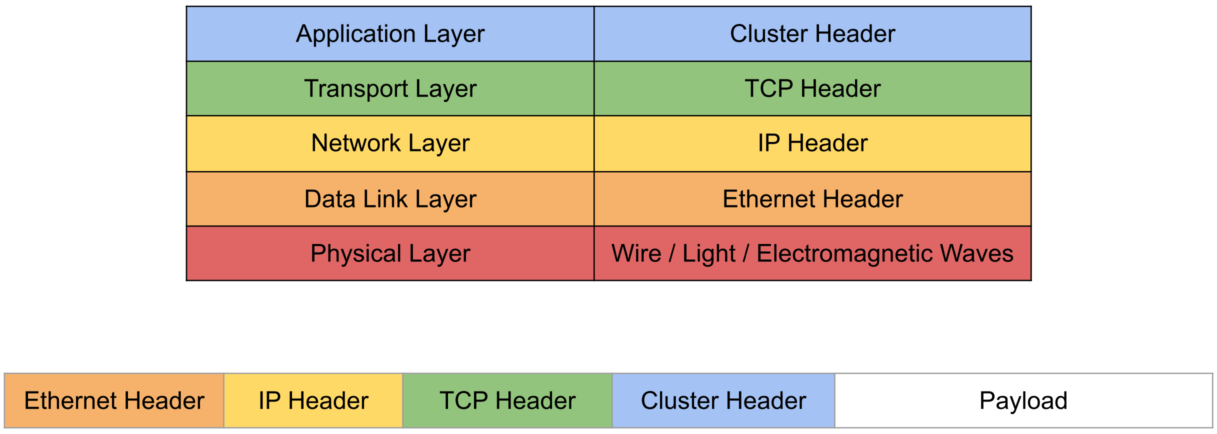
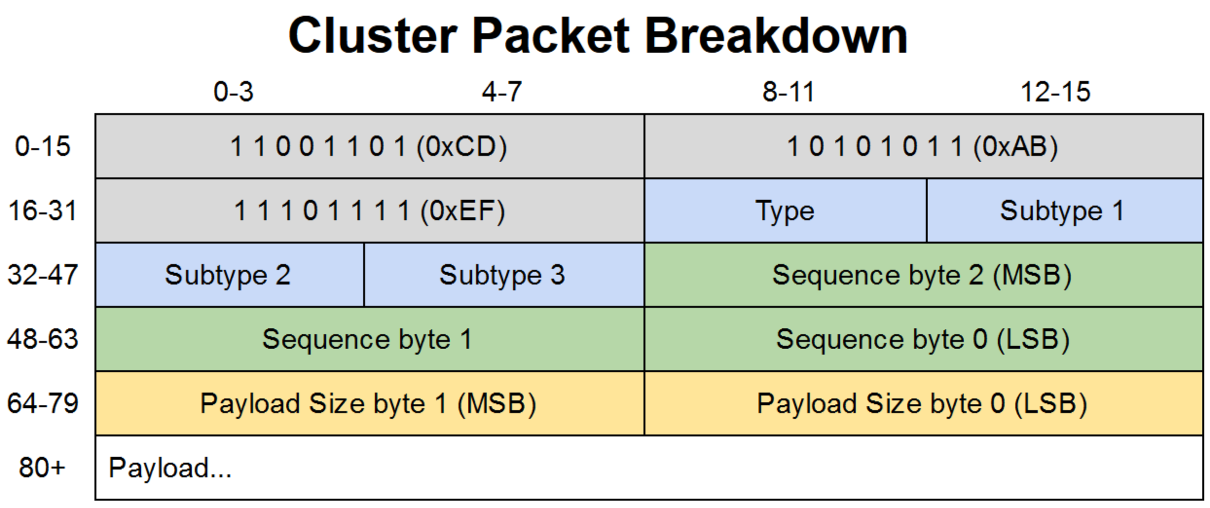


Figure 3.4: Cluster OSI Stack

The five required layers of the OSI stack for all computer networks are shown in Figure 3.4. Once a program tells the network hardware of its host machine to send the data, the TCP header is added to allow data verification, network speed control, and organization of multi-packet file transfer. Then the Internet Protocol (IP) header is added to mark the IP addresses of the source and destination. Finally the Ethernet header is added to identify the hardware that sent the data and that will receive it so things like Ethernet switches can optimize the packet's path through the network. The addition of the Cluster header is done in the application layer by the Cluster program, both mobile and server.

A Cluster packet, diagrammed in Figure 3.5, is any packet of data sent between devices in the Cluster network, including the server. A Cluster packet must have the Cluster header added in its application layer before the payload bytes. The program checks the validity of each packet's Cluster header before



Cluster Packet Breakdown

	0-3	4-7	8-11	12-15
0-15	1 1 0 0 1 1 0 1 (0xCD)		1 0 1 0 1 0 1 1 (0xAB)	
16-31	1 1 1 0 1 1 1 1 (0xEF)		Type	Subtype 1
32-47	Subtype 2	Subtype 3	Sequence byte 2 (MSB)	
48-63	Sequence byte 1		Sequence byte 0 (LSB)	
64-79	Payload Size byte 1 (MSB)		Payload Size byte 0 (LSB)	
80+	Payload...			

Figure 3.5: Cluster Packet Header

interpreting what to do with it.

The Cluster header starts with three bytes that are always the same series of bits containing both ones and zeros in a specific order that are unlikely to occur by accident in this specific grouping; the bits in hexadecimal (hex) are 0xCDABEF. The next two bytes (four 4-bit fields, bits 24 – 39) identify what type of Cluster packet it is, meaning what kind of data or command it contains. This is broken down into 4 subtypes which are discussed in section 3.7. The next three bytes (bits 40 – 63) contain space for a 24-bit number, typically used to number the packets involved in a file transfer so the receiving program knows that all file fragments arrived in order. The last two bytes (bits 64 – 79) are a 16-bit integer specifying how many bytes the payload is. The payload may be up to 1400 bytes (the limit of an Ethernet packet).

3.7 Cluster Packet Types

Tables 3.1 – 3.12 show the defined values for each of the subtype fields in the Cluster Header, see Figure 3.5 for the byte structure of the Cluster Header. The client and server programs reject any packet whose type fields contain invalid values or combinations of subtypes.

Packet Type Bits	**Packet Type**
0x0	Ping
0x1	Task Data
0x2	ACK
0x8	Distribution Update
0xA	Cluster Update

Table 3.1: Packet Type

Table 3.1 shows the possible valid values of the Type field for the Cluster Header. The ping type is for confirming that the device is still connected and running the Cluster app (see Table 3.2). Task Data indicates a packet whose payload includes data specific to the current computation task (see Table 3.3). ACK is an acknowledgment that the previous packet was received and may indicate success or failure (see Table 3.4). A Distribution Update is a change in the number of devices, the indexing of the devices, or other event that affects how the computation load will be distributed (see Table 3.5). A Cluster Update contains a command that changes the task running, changes the state of a task, or another command related to the running task (see Table 3.6).

Ping Bits (S1)	**Ping Type**
0x0	Hi (Ping)
0x1	Task Running
0x2	Task Paused

Table 3.2: Packet Subtype 1: Ping Commands

Subtype 1 breaks down the packet type into more specific instructions. Table 3.2 shows the possible subtypes for the Ping type. The “Hi” ping is what the server sends to the devices. The response from the device indicates whether the task is still running.

Task Data Bits (S1)	**Task Data Type**
0x0	Generic
0x1	Large File
0x2	Text (ASCII)
0x3	Numeric
0x8	Request
0xA	Shared

Table 3.3: Packet Subtype 1: Task Data Packets

Table 3.3 shows the possible subtypes for the Task Data type. The Large File type is for transferring whole files over multiple packets, this uses the Sequence bits shown in Figure 3.5. The Text type is for generic text encoded as ASCII. The numeric type is for storing numbers either as text or actual integers and floating point numbers stored as a predictable representation according to IEEE standards. The Request type is for requesting data, the payload of which may contain more details about the request. The Shared type is for transferring data that is to be shared with other devices, such as for a dependent calculation, see Section 4.2.2.

ACK Bits (S1)	**Ack Type**
0x1	Acknowledged
0x2	Acknowledged and Completed
0x3	Completed
0xF	Error Condition

Table 3.4: Packet Subtype 1: Acknowledgments

Table 3.4 shows the possible subtypes for the ACK type. The ACK type is subdivided into 4 conditions that indicate (in order): (1) a general acknowledgement, (2) ACK and completed (success), (3) completed, a success condition that can be sent asynchronously after the initial ACK if the action may take time to complete, and (4) negative ACK that indicates an error state.

Distribution Update Bits (S1)	**Distribution Update Type**
0x0	Client Event
0x3	Client Command
0x6	New Number of Devices
0xF	New Index

Table 3.5: Packet Subtype 1: Distribution Updates Commands

Table 3.5 shows the possible subtypes for the Distribution Update type. A client event may be any general announcement from the devices, such as overheating, a need to enter idle (rest) mode, or other errors. Client commands are commands sent by a mobile device for other client nodes or the server node. The other two update commands are for telling the client nodes that there are now a different number of devices or there is a change in the numbering of the devices or the distribution of the work.

Cluster Update Bits (S1)	**Cluster Update Type**
0x0	Task Command
0x5	Update GUI
0xD	Database Command
0xF	Task Change

Table 3.6: Packet Subtype 1: Cluster Update Commands

Table 3.6 shows the possible subtypes for the Cluster Update command. This includes general task commands (see Table 3.9), a request to update the app's GUI (graphical user interface), a database-specific command (see Table 3.10), or a change in which task the cluster is working on (see Table 3.9).

File Packet Bits (S2)	**File Packet Type**
0x1	File Data
0x2	File Metadata

Table 3.7: Packet Subtype 2: File Packets

Table 3.7 shows the possible subtypes for the File type packets. This is either file metadata, which is sent first before the contents of the file during a large file transfer, or the data itself divided into fragments and organized (ordered) using the Sequence bytes in the Cluster packet header.

Numeric Task Data Bits (S2)	**Numeric Type**
0x1	Integer
0x2	Floating Point
0x4	Integer Text
0x7	Floating Point Text
0x9	Generic Text

Table 3.8: Packet Subtype 2: Numerical Data Types

Table 3.8 shows the possible numerical data types that the receiving device should expect when reading a Task Data packet. These include integers and floating-point numbers encoded in 32-bit or 64-bit groups as defined by IEEE standards. This would be a more efficient way of encoding lots of numeric data if neccessary, or the numbers could be simply ASCII text representations of decimals or integers.

Task Command Bits (S2)	**Command**
0x1	Start
0x2	Stop
0x5	Change Stage

Table 3.9: Packet Subtype 2: Task Commands

Table 3.9 shows the possible task commands. These include starting the task, pausing or starting the task, or changing the stage of the task currently running. The stage change applies to only those tasks that have multiple stages; only one has been implemented so far, the FAWN key-value data store (Section 4.1.2).

Database Command Bits (S2)	**Database Command**
0x1	Clear All
0x2	Clear Data

Table 3.10: Packet Subtype 2: Database Commands

Table 3.10 shows the possible database commands for the FAWN database. See Section 4.1.2. Clear All erases all data and keys in the database. A Clear Data command clears only the actual data stored at each of the keys in the database, but does not erase any of the keys.

Database Store Command Bits (S3)	**Store Command**
0x01	Store
0x02	Fetch

Table 3.11: Packet Subtype 3: Database Store Commands

Table 3.11 shows the possible database store commands. This is the only subtype so far to occupy the Subtype3 field, except for the Task ID (or Task Change) subtype, which occupies both the Subtype 2 and 3 fields together. The store commands are only used for the FAWN data store task and tell the device to either store data along with an accompanying key or to retrieve data with a certain key.

Task Change Bits (S2:S3)	**Task Type**
0x00	Nothing
0x01	Python Script
0x02	OpenCV Script
0xC1	Synthetic: Server Prep
0xC1	Synthetic: Dependent Calc
0xE0	HPC Demo: Digits of Pi
0xE1	CPS Demo: Image Prep
0xE2	CPS Demo: Matrix Math
0xE3	CPS Demo: Inverse Kinematics
0xE4	CPS Demo: FAWN

Table 3.12: Packet Subtype 2 & 3: Task ID

Table 3.12 shows the possible tasks implemented in the Cluster app. The Task ID types show all the currently implemented tasks and a few planned tasks to be designed and implemented in the future. The application tasks are examples of tasks that might be used in an actual user implementation of the Cluster network, for demonstrating how common computing tasks for cyber-physical systems (CPS) or general-purpose high-performance computing (HPC) might be accomplished with a group of retired smart devices. The synthetic tasks were designed and programmed to characterize how certain types of tasks would perform in the Cluster without implementing a real-world algorithm such as Monte Carlo simulations or physics models, but instead are carefully controlled to behave a particular way, such as creating a compute bottleneck at the server compute (see Section 4.2.1), or enforcing that the client nodes depend on data from partial calculations done by the neighboring nodes in order to continue their calculations (see Section 4.2.2).

3.8 Structure of the Cluster Software

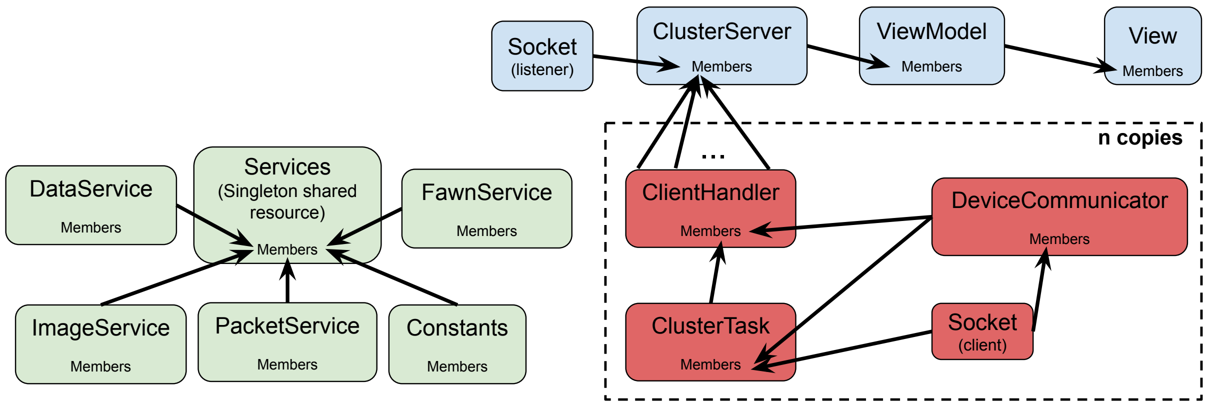


Figure 3.6: Cluster Server Class Hierarchy

The classes within the Cluster program on both mobile device and server were designed in a hierarchy so that the resources allocated for this program, as the number of clients grows, would include a minimal number of class instances. Not all classes need to have a new instance per connected device. This is particularly important on the server node where the the amount of work the processor is doing grows with the number of devices it is managing. The class hierarchy for the server program, written in C#, is shown in Figure 3.6. The server program conforms to the Model View ViewModel (MVVM) structure of agile software development. This allows for organization, abstraction of which classes can access which resources, and ease of reuse since the layers of the hierarchy are not bound to each other and the designed classes can be used as-is for other projects without rewriting them.

The only classes that need to have a new instance per connected device are shown in red in Figure 3.6. The ClientHandler class keeps track of which task is being done, and information about the connected device and the socket connection to the device. This socket is governed by the DeviceCommunicator class which is the only class allowed to talk to the device and which also handles adding on the Cluster header before data is sent and interpreting all data recieved from the client device. Finally the ClusterTask class is an abstract class which has several subclasses for specific tasks that the device cluster can do. This class handles gathering resources and passing messages that one particular device will need to do its work.

Services are shared classes and libraries that are accessible globally within the the server program. The Data Service prepares arrays of synthetic data on demand at run-time using random number generation. This is especially useful for the synthetic tasks to simulate intensive computations or the collecting of data from sensors or databases, but is also used by the FAWN task. The Packet Service is a library of functions that help standardize Cluster packets of all types and append the cluster header. The Constants service is a collection of global constants for organization and easier updating during code redesign or reuse. The Image Service is used by the ImagePrep task to fetch and track the status of images from the server computer's hard drive. The FAWN Service is used by the FAWN task to generate and track each of the key-value pairs sent and later queried from the client nodes.

A Cluster task can be one of several types of operations, from just maintaining the client connections with a simple pinging operation to other more complicated computing tasks. The structure of the Cluster task hierarchy is shown in Figure 3.7.

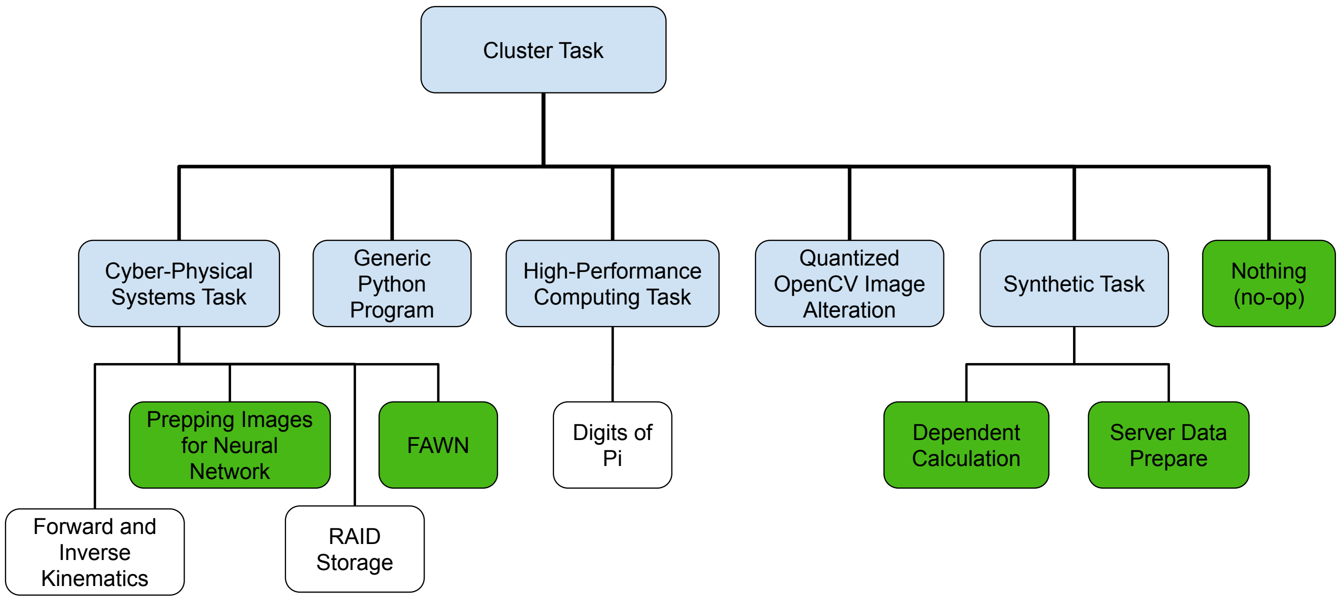

Cluster Task
Cyber-Physical Systems Task
Generic Python Program
High-Performance Computing Task
Quantized OpenCV Image Alteration
Synthetic Task
Nothing (no-op)
Prepping Images for Neural Network
FAWN
Digits of Pi
Dependent Calculation
Server Data Prepare
Forward and Inverse Kinematics
RAID Storage

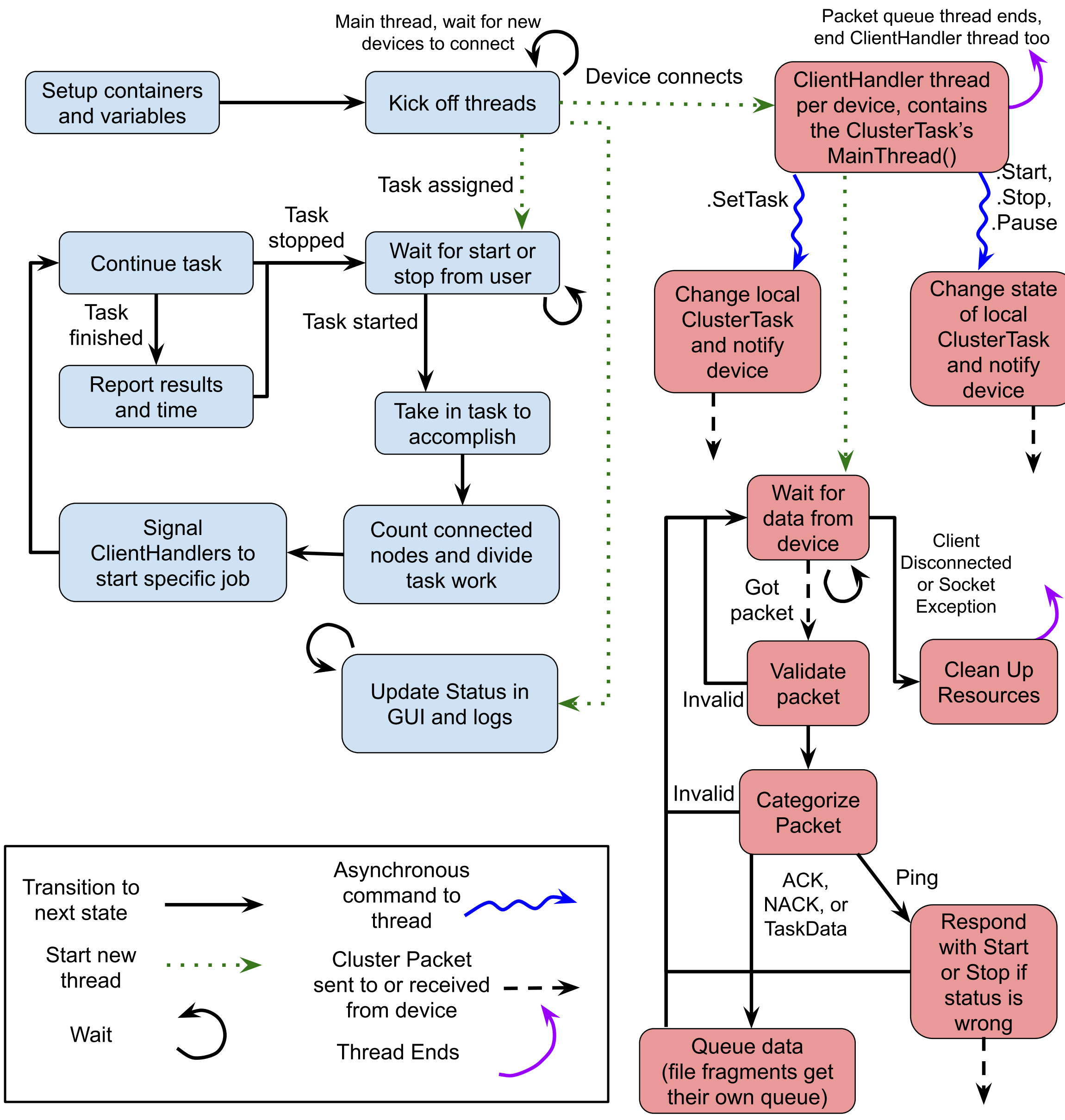
Figure 3.7: Cluster Task Hierarchy

Five of the tasks, shown in green, that are depicted in the hierarchy were implemented as part of this work: two of the CPS tasks (ImagePrep and FAWN), both of the synthetic tasks, and the Nothing (no-op) task. The CPS tasks demonstrate preparing images for a neural network or implementing the FAWN data store to test read and write speed of a FAWN implementation (see Section 4.1.2). The synthetic tasks were designed to identify bottlenecks in the scalability of the Cluster network by simulating large workloads in various places such as on the server computer or during the wait time when other dependent nodes are preparing and passing on their contributions to partially-completed calculations.

The other tasks were theorized and will likely be implemented and characterized in future work. The Digits of Pi task would be an implementaion of the classic parallel processing algorithm for calculating digits of pi, an application similar in profile to the synthetic DependentCalc task. The Forward and Reverse Kinematics task would implement the linear algebra used by the robot arm discussed in Section 1.2.1. The RAID Storage task would implement the RAID storage architecture of redundant and interleaved data backups, similar to the FAWN data store, but much more generic; the RAID task would be able to store files and even whole filesystems instead of individual values associated with keys. The Generic Python Program is a theoretical applicaton for Cluster where the client nodes may be able to run any Python program meeting certain criteria and perform parallel processing using techniques similar to MPI or even using an MPI library. The Generic Python Program task would support a Function-as-a-Service (FaaS) model for the cluster similar to Renée [27] (see Section 5.8). The Quantized OpenCV Image Alteration task would be similar to the ImagePrep task and would accept an image and a set of commands that translate to OpenCV operations to be performed to alter the image before sending it back.

The program flow of the Cluster Server program is illustrated in Figure 3.8. The C# server program has $3 + 2n$ threads: the main thread to handle incoming connection requests and spawn new threads for each of them, a thread that responds to commands from the user interface, a thread to update the GUI with the latest status from the server object, and two threads per connected device. The first of these two handles all incoming packets from the device (DeviceCommunicator thread) and either routes them to a packet queue or deals with them immediately if the packet payload is a command from the server not directly related to the task being run. Incoming packets from the connected devices are validated and queued in this dedicated thread, consumed only when the task's main thread needs them. Incoming file fragments (identified by part of the Cluster packet header) are stored in their own queue for easier access by the task's MainThread.

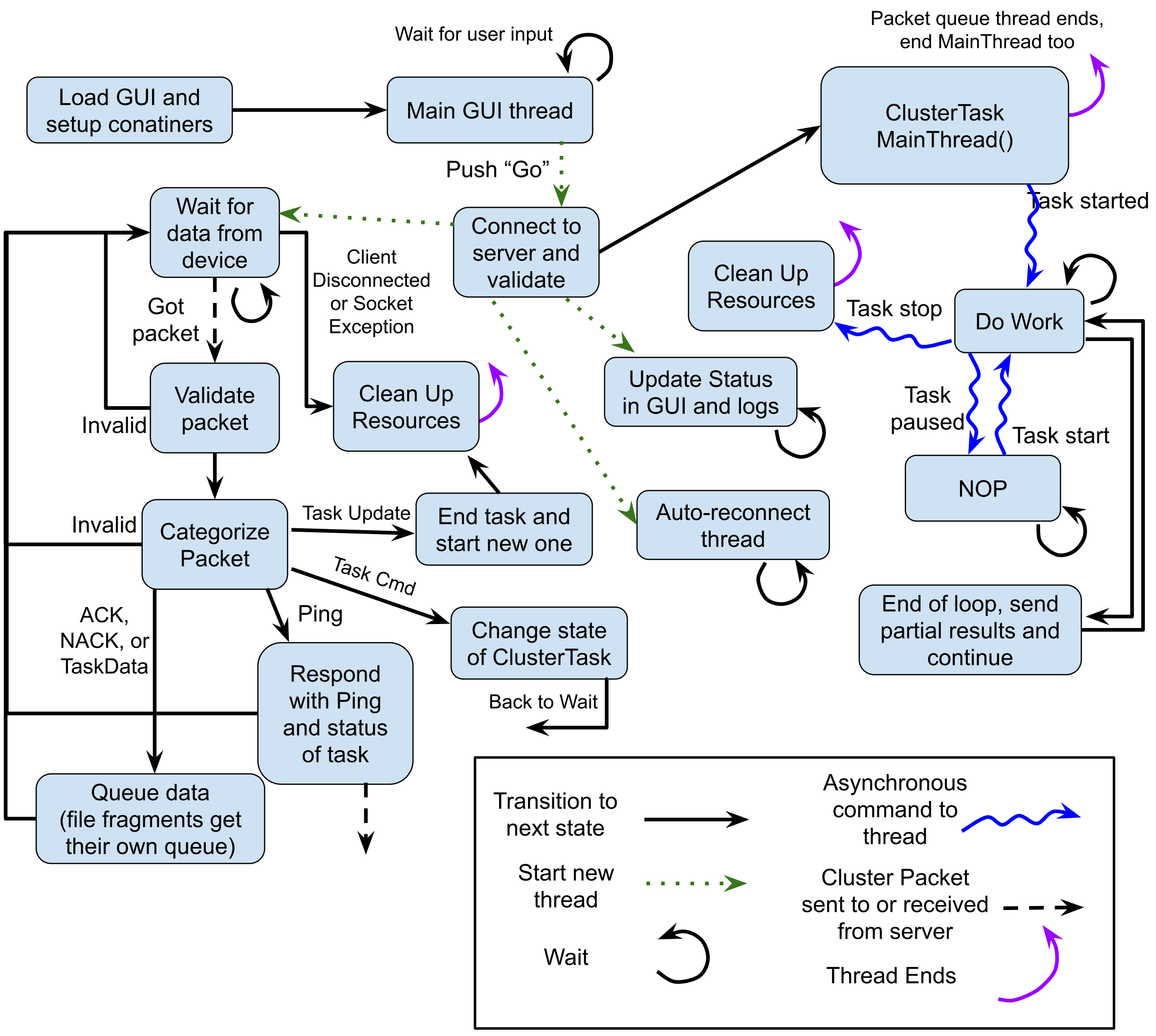
Some incoming packets are status pings or other special packets that are handled by the DeviceCommunicator thread and not queued. The second of those connected device threads is the ClusterTask's MainThread() to manage the different steps of the task itself and cooperate with the client node assigned to that thread to accomplish the task.



Main thread, wait for new devices to connect
Setup containers and variables
Kick off threads
Device connects
Packet queue thread ends, end ClientHandler thread too
ClientHandler thread per device, contains the ClusterTask's MainThread()
Task assigned
Task stopped
Continue task
Wait for start or stop from user
.SetTask
.Start, .Stop, .Pause
Change local ClusterTask and notify device
Change state of local ClusterTask and notify device
Task finished
Task started
Report results and time
Take in task to accomplish
Signal ClientHandlers to start specific job
Count connected nodes and divide task work
Wait for data from device
Client Disconnected or Socket Exception
Got packet
Update Status in GUI and logs
Validate packet
Invalid
Clean Up Resources
Invalid
Categorize Packet
ACK, NACK, or TaskData
Ping
Respond with Start or Stop if status is wrong
Queue data (file fragments get their own queue)
Transition to next state
Start new thread
Wait
Asynchronous command to thread
Cluster Packet sent to or received from device
Thread Ends

Figure 3.8: Cluster Server Program Flow

The program flow of the Cluster app, which runs on each node, is illustrated in Figure 3.9. The app has five



Wait for user input
Packet queue thread ends, end MainThread too
Load GUI and setup conatiners
Main GUI thread
ClusterTask MainThread()
Push "Go"
Task started
Wait for data from device
Connect to server and validate
Client Disconnected or Socket Exception
Clean Up Resources
Task stop
Do Work
Got packet
Update Status in GUI and logs
Invalid
Validate packet
Clean Up Resources
Task paused
Task start
NOP
Invalid
Categorize Packet
Task Update
End task and start new one
Auto-reconnect thread
Task Cmd
End of loop, send partial results and continue
ACK, NACK, or TaskData
Ping
Change state of ClusterTask
Respond with Ping and status of task
Back to Wait
Queue data (file fragments get their own queue)
Transition to next state
Asynchronous command to thread
Start new thread
Cluster Packet sent to or received from server
Wait
Thread Ends

Figure 3.9: Cluster Node Program Flow

threads: the main thread that responds to commands from the user interface, a thread to update the graphical user interface (GUI) with the latest status and image (if applicable) from the task, a thread to automatically attempt to reconnect to the server if the socket becomes disconnected, one to handle all incoming packets from the server (Listener thread), and the ClusterTask's MainThread() to complete the various steps of the task itself and send out incremental or finished results. Incoming packets from the server are validated and queued in exactly the same way as with the server program with some additional special packets. Upon receiving a ping, the app responds with another ping and the status of the current task. Upon receiving special commands for starting, stopping, pausing, or switching the task, those packets are dealt with immediately and not queued.

3.9 Multi-Process Python Cluster Server Program

To test research question number 2 (Section 3.3) regarding a multi-process approach to the server program, a second version of the program was created. This version is written in Python because C# and Visual Studio have very little support for multi-process programs. Python has convenient stable libraries for multi-process programming and communication between processes, which is also possible in C#, but with considerably more work required for the programmer. A Python program also has the added benefit of being cross-platform, so it can run on almost any modern computer. Note that while the Python program has been confirmed to run on MacOS and Linux operating systems, it has not been thoroughly tested or measured on those operating systems.

The Python version of the program is multi-processed in addition to multi-threaded. Multi-threaded programs can use only one processor core of the computer they run on. All of the memory and processor resources are limited to what a single process is allowed to have. Multi-process programs divide work among several processes, which means the processor can actually work on the task in several cores simultaneously. Large programs that need to do many simultaneous tasks, like Google Chrome [76], use multi-process design. Each process has its own allocated memory space on the processor. However, a multi-process design takes significant time to create all the processes, but is equal to a multi-threaded approach in terms of context switching time within one core after the processes are running. This equality is because all threads running on a single core function exactly the same as typical multi-threaded programs of only one process [77, 78, 79].

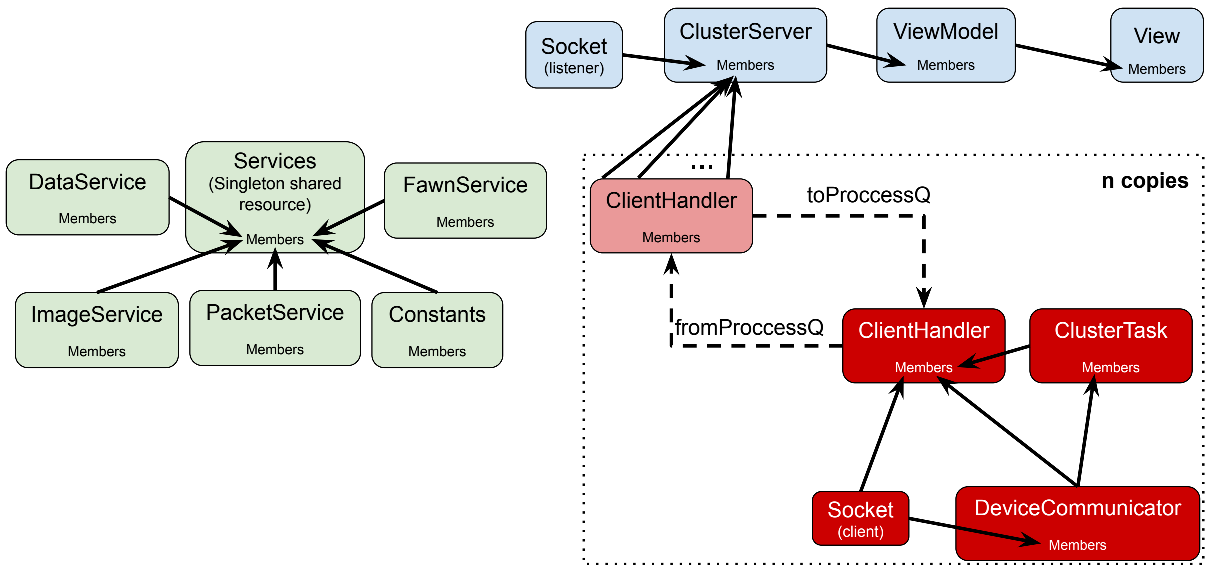


Figure 3.10: Multi-Process Cluster Server Class Hierarchy

The structure of the Python implementation of the Cluster server is identical to the C# version in many aspects, with a few differences related to how computer resources are shared, and how the program is divided when new devices connect, and how global information about the state of the task is communicated between threads. Threads within a process all share one memory space and therefore can access global data structures that track the state of the running task. However, processes, even those belonging to a single program, all have their own distinct memory space that is not shared. In order for all processes of a multi-process program to stay up to date about the state of the task, messages must be passed between processes through pipes. This is similar to how nodes of a distributed computer would pass messages using MPI. First, Figure 3.10 shows the class hierarchy in which the client handler instances fork a new process and then creates the ClusterTask and DeviceCommunicator instances only on the new process.

The program flow for the Python server is illustrated in Figure 3.11. There are $3 + n$ threads on the main process, and n additional processes each with their own three threads. The n threads on the main process are very light weight compared to the $2n$ threads on the C# server. They only wait for a process to request a shared resource and provide it when requested. The three threads within the individual processes are responsible for listening for messages from the main process, listening for data from the mobile device (client node), and the MainThread() which coordinates with the task being done on the client node.

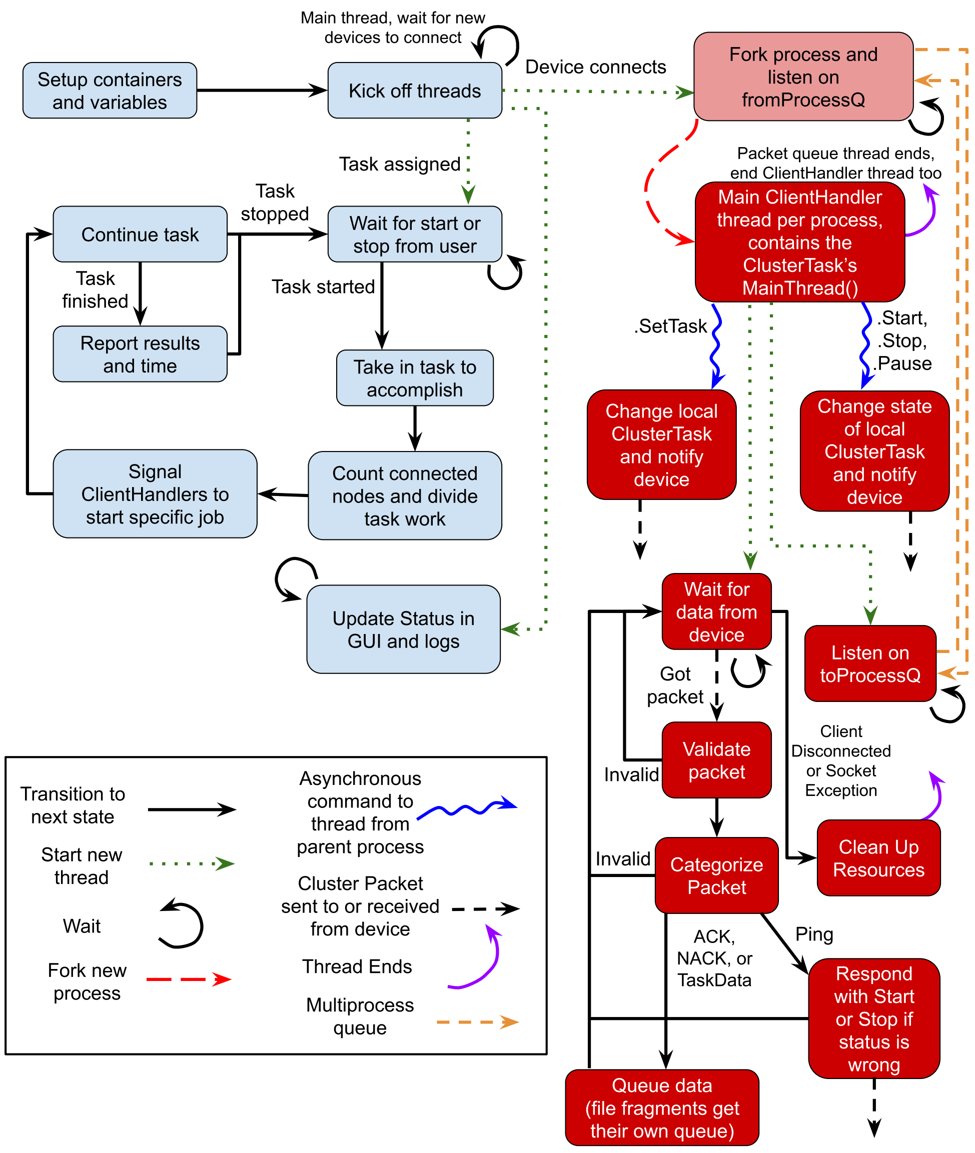

Main thread, wait for new devices to connect
Setup containers and variables
Kick off threads
Device connects
Fork process and listen on fromProcessQ
Task assigned
Packet queue thread ends, end ClientHandler thread too
Task stopped
Continue task
Wait for start or stop from user
Main ClientHandler thread per process, contains the ClusterTask's MainThread()
Task finished
Task started
Report results and time
Take in task to accomplish
.SetTask
.Start, .Stop, .Pause
Change local ClusterTask and notify device
Change state of local ClusterTask and notify device
Signal ClientHandlers to start specific job
Count connected nodes and divide task work
Update Status in GUI and logs
Wait for data from device
Listen on toProcessQ
Got packet
Validate packet
Invalid
Client Disconnected or Socket Exception
Invalid
Categorize Packet
Clean Up Resources
ACK, NACK, or TaskData
Ping
Respond with Start or Stop if status is wrong
Queue data (file fragments get their own queue)
Transition to next state
Start new thread
Wait
Fork new process
Asynchronous command to thread from parent process
Cluster Packet sent to or received from device
Thread Ends
Multiprocess queue

Figure 3.11: Multi-Process Cluster Server Program Structure

3.10 Software Metrics

With two versions of the Cluster server program and two versions of the Cluster app, comparing the size of each of the pieces of Cluster software is an important metric in terms of resources needed for each choice of implementation. Table 3.13 shows the size of the code base of each program as well as the size of the executables.

Program	Lines of Code	Size of Executable
iOS Cluster App	6974	6.6 – 8.1 MB
Android Cluster App	3564	34.85 – 37.93 MB
Python Multi-Process Server Program	3683	13.781 MB
C# Threaded Server Program	6075	398.01 kB

Table 3.13: App and Server Program Metrics

For time reasons, after the initial ImagePrep experiments comparing threaded vs. multi-process, see Section 5.1, the implementation effort of the server program were focused on the Python program for the potential benefits of its parallel processing. Although the multi-process Python server program has more features than the C# program, the Python program has only 60.6% of the lines of code of the C# program, while the executable is 34.6 times larger. However, the size of the Python executable is irrelevant to this research as the experiments were only run with the Python interpreted in real-time and not compiled.

The iOS app similarly has one more feature than the Android app, specifically the FAWN data store implementation, and yet is 1.96 times the size of the Android app in terms of lines of code, and 20.2% the size of the Android app on a smart device's hard drive on average.

3.11 Hardware Selection

Because Cluster is designed to function with any iOS or Android mobile device, the implementation was tested with a variety of different makes and models. This heterogeneity is one important distinguishing characteristic of Cluster compared to related works. A wide range of devices was acquired for characterizing the implementation of Cluster. A total of 25 used phones and tablets were donated to the cause (not all of which ended up being used), see section A.2.1, and six used devices were purchased from eBay. The device model introduction years range from 2012 to 2017. Below is a list of all available devices used in various experiments throughout this work.

#	Model	Year	Operating System	Processor	Storage	Memory
1	iPhone XS	2017	iOS 16.5	2-core 2.5 GHz Vortex + 4-core 1.6 GHz Tempest	64 GB	4 GB
7	iPhone 6s	2015	iOS 15.7	2-core 1.84 GHz Twister	16 – 64 GB	2 GB
1	iPhone 6	2014	iOS 12.5.6	2-core 1.4 GHz Typhoon	64 GB	1 GB
1	iPhone 5s	2013	iOS 12.5.6	2-core 1.3 GHz Cyclone	16 GB	1 GB
4	iPhone 5c	2013	iOS 10.3.3	2-core 1.3 GHz Swift	16 GB	1 GB
1	iPad mini 2	2013	iOS 12.5.6	2-core 1.3 GHz Cyclone	16 GB	1 GB
2	iPad 4	2012	iOS 10.3.3	2-core 1.4 GHz Swift	16 GB	1 GB

(a) iOS

#	Model	Year	Operating System	Processor	Storage	Memory
1	Galaxy S8	2017	Android 9.0 Pie	4-core 2.35 GHz Kryo + 4-core 1.9 GHz Kryo	64 GB	4 GB
1	Galaxy S6	2015	Android 7.0 Oreo	4-core 2.1 GHz Cortex + 4-core 1.5 GHz Cortex	32 GB	3 GB
1	Nexus 5x	2015	Android 8.1 Oreo	4-core 1.4 GHz Cortex + 2-core 1.8 GHz Cortex	32 GB	2 GB
5	Galaxy S5	2014	Android 6.0 Marsh.	4-core 2.5 GHz Krait 400	16 GB	2 GB
1	Nexus 5	2013	Android 6.0 Marsh.	4-core 2.3 GHz Krait 400	16 GB	2 GB
1	Galaxy Tab E	2016	Android 7.1 Nougat	4-core 1.3 GHz Cortex-A53	16 GB	1.5 GB

(b) Android

Table 3.14: List of Mobile Devices Available for Use in Cluster

Other hardware selected for the experiments in this research are as follows:

1. The server computer: an ASUS ROG GL752V gaming laptop, Windows 10, 1 TB Samsung Evo SSD, 16 GB RAM
2. The Wi-Fi router (for the first three experiments comparing ImagePrep performance for different server types): a NETGEAR N300 EX2700 Wi-Fi repeater
3. The Wi-Fi router (for all other experiments): a Rockspace AC750 Wi-Fi repeater
4. The network switch (which connected the Wi-Fi router to the server computer): TP-Link TL-SG105, 5-Port Gigabit Ethernet Switch
5. CAT6-rated Ethernet cables for connecting all wired network equipment

For all experiments, the server computer's Wi-Fi was switched off and the private network that the experiments were run on was isolated from the internet. The Wi-Fi network operated exclusively on the 2.4 GHz band for all experiments.

Chapter 4

Cluster Characterization Task Implementation

This chapter discusses implementations of the four tasks that were used to characterize how well Cluster performs at various types of computations. To characterize the performance of the device cluster, many experiments needed to be done, recording the metrics for completing certain computing tasks. The environment for the characterization experiments was set up as follows: For each experiment the mobile devices were laid out on 2 inches of corrugated cardboard sitting on top of 4 inches of cushion foam. That assembly was elevated 20 centimeters off the floor by rigid, hollow, rectangular, cardboard brackets. The mobile devices were all plugged into power via their charging cables at all times and 3 centimeters of clearance was maintained such that no device and no cable were within 3 cm of another device. The use of non-conductive materials and spacing of devices was intended to establish a baseline wireless environment for characterizing the Cluster network performance.

4.1 Cluster Cyber-Physical System Tasks

Cyber-Physical System tasks are fully-implemented computation tasks that are important to realizing specific applications for cyber-physical systems. Two such tasks were implemented so far out of many that were theorized to be possibly suitable for Cluster. These include pre-processing images for a neural network and implementing the FAWN key-value data store to compare read and write speed of a FAWN implementation on retired mobile devices to the original paper's cluster of "wimpy nodes" performance [28, 29].

4.1.1 Image Preparation Task

The ImagePrep task was designed to be an independent task in which the client nodes receive an image from the server, perform several operations on the image, and send it back. This the simplest possible type of non-trivial task since there is no dependency between nodes and the distribution of work is not critical; any image can be assigned to any node for processing or reassigned at any time. This also means that each client node is completely hot-swappable, i.e. can be removed from the network or added to the network at any time, including while the task is in-progress.

4.1.1.1 Motivation

There has been significant research done and many papers published [80, 81, 82] on the benefits of pre-processing images either as a top layer in a convolutional neural network or as a computation task applied before the neural network receives the pre-processed images as input data. The idea of doing the pre-processing before the neural network receives the images is that the layered network could be simplified, and therefore process image data faster and with lower power, if the neural network doesn't need to do the pre-processing work itself.

While this task for Cluster does not reproduce the exact pre-processing algorithm from any published work, it performs many of the same operations and serves as a proof-of-concept for how an image pre-processing task would perform on Cluster. This type of task has applications beyond neural networks as well. Any CPS application that requires many images to be collected and then stored, such as analyzing images from space telescopes [83, 84, 85], could use a task like this to standardize the images before they are stored or used in further steps of the application. Beyond image pre-processing, other "embarassingly parallel" tasks, such as processing the 1.25 GB of data that is created every second at the ALICE project of the Large Hadron Collider [86], are expected to have similar performance.

4.1.1.2 Operation

On startup of the server program, before any nodes have connected, the server locates all images available in a predetermined directory (a test data set of 2000 images of varying size and aspect ratio were used in these experiments) and stores the paths in an array. Then, after the client nodes connect and the task begins, each client node requests an image to process. The server sends the file in fragments, tracking each fragment using the sequence number in the Cluster header (Figure 3.5), and re-sending any packets that were corrupted or missing. The client acknowledges the successful transfer of the file, performs the alterations on the image, and sends the image back with similar file fragmentation and confirmation of file transfer.

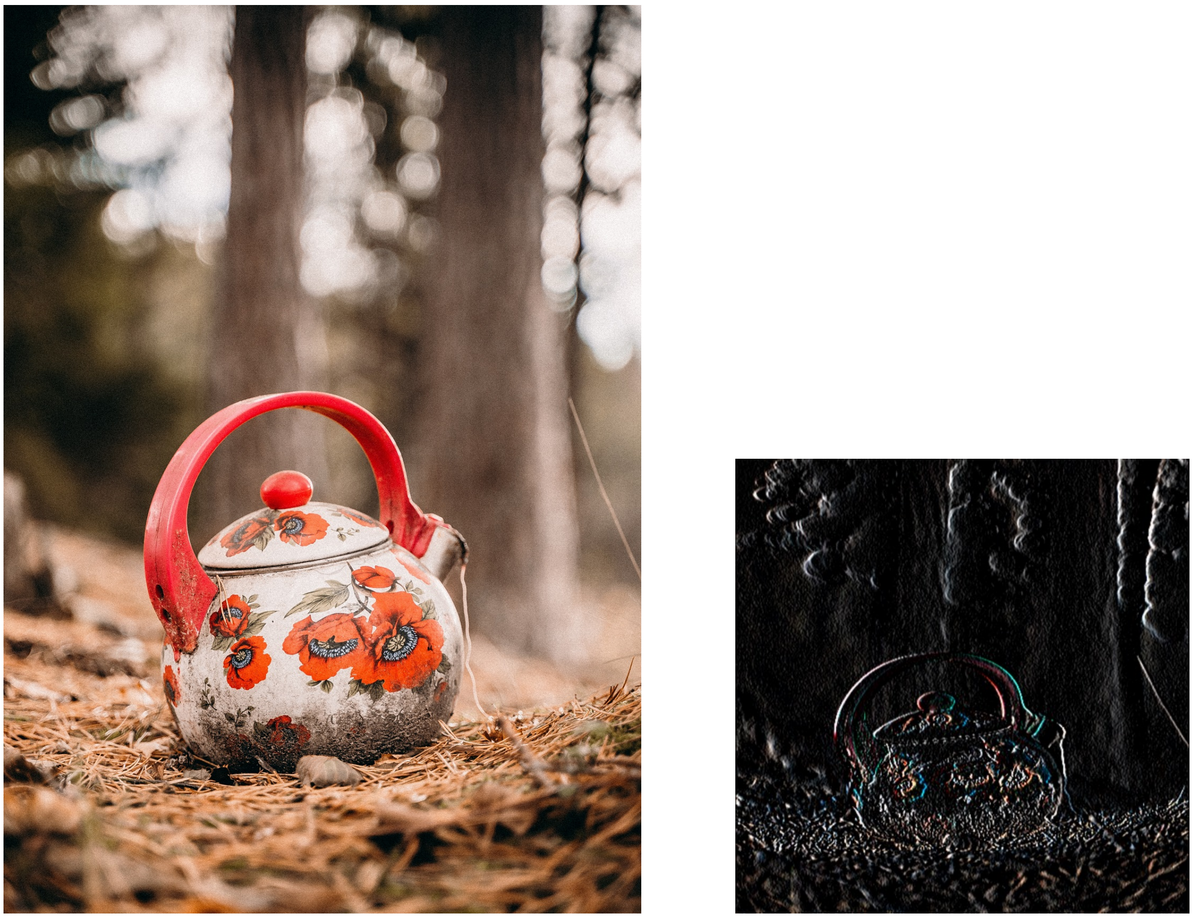

(a) (b)

Figure 4.1: (a) Original image. (b) Processed image.

The operations are typical of pre-processing that might be done to images to be inferred by a neural network that was also trained on similarly prepared images. This kind of neural network can have reduced complexity in its architecture because the image resizing and other operations don't need to be done by the network itself. The full list of operations done on each image for this task is shown below:

1. Resize image to 500 × 500 pixels, stretching as needed, not cropping.
2. Reduce color saturation to half of original levels.
3. Apply Sobel edge detection (dx=1, dy=0).
4. Apply Gaussian blur to entire image (0.5 radius).

The criteria for task completion is when all 2000 images have been successfully returned to the server in an altered state and verified as a valid PNG file. An example of an image used in this task is shown to scale in Figures 4.1a (before processing) and 4.1b (after processing).

4.1.1.3 Characterization

Every task is characterized by measuring its running time, network traffic, and power usage. Each characterization experiment was run at least 20 times per data point. So each recorded data point includes the mean and standard deviation of 20 trials of the experiment. Additional task-specific parameters for each run are recorded during the characterization experiments as well. The data points are taken for an increasing number of connected devices to understand how Cluster's performance on this highly parallel task scales. The parameters and statistics recorded for each individual trial of the ImagePrep experiments are listed below:

1. Timestamp
2. Number of client nodes
3. Total number of images processed (2000 for the experimental data set)
4. NetworkReceived: total kilobytes received from the client nodes
5. NetworkSent: total kilobytes sent to the client nodes
6. TaskData: total kilobytes of images to be processed
7. MainThread loop delay (100 ms)
8. Measured elapsed time to complete the task

Note that the network data sent and received statistics include all traffic including acknowledgments and task commands, and not just the image file fragment packets. These metrics were recorded to be able to measure how much network traffic is required compared to the size of the data set.

The MainThread loop delay is how much idle time the server node waits for one particular client connection between receiving the processed image and sending another. The delay was added to give the client nodes a chance to idle or service background tasks run by the mobile OS and synchronize with the corresponding server thread that is also transmitting files. It was observed during system development that the Cluster app crashed more often and the ImagePrep task took much longer to finish without any delay between processing one image and the next. The cause is twofold. First, traditional mobile CPU throttling (clock frequency reduction) techniques that aim to maximize utilization of the available thermal headroom can cause the mobile device to initially heat up from too much continuous processing; the subsequent reduction in clock frequency to compensate for heating leads to an overall increase in execution time as shown by Sahin and Coskun [87]. Sahin and Coskun propose reducing the maximum allowable clock frequency or idling cores on a mobile multicore processor to increase the time duration that performance is sustained by the mobile device. The added delay in the Cluster server program achieves a similar function, namely load balancing for the client nodes to maintain sustained performance on the Cluster tasks. Note that operating a device at elevated temperature can also cause errors in the execution of any program due to transient hardware faults induced by thermal noise, timing violations due to circuit delay variations, and power supply voltage droop.

Second, errors occur while transmitting data wirelessly across a network and occasionally the Cluster server does not receive the acknowledgment from the client node that the end of the file was received. When this happens, the server node is stuck waiting for the acknowledgment to arrive even though the image processing proceeds on the client side. It is difficult to synchronize two programs on two different computers that are each sending and receiving a large of amount of data repeatedly at various stages of the program. Despite a complex algorithmic design that relies on acknowledgments to move on to the next stage of the program, when errors in transmission occur, the programs can become unsynchronized and therefore some packets are sent across the network that are ignored by the receiving program, which is expecting a packet with a different payload. Without the added delay, the transmitter and receiver have no chance to recover from these misalignments or take significantly longer to re-synchronize. The MainThread loop delay is currently programmed as 100 milliseconds for the ImagePrep task. Further study on the effects of the server delay and client node idle time on system performance is planned for the future, see Section 6.3.

The ImagePrep task was the first task implemented as part of this research and it has been the most thoroughly characterized and implemented in several variants to test the performance effects of varying the electromagnetic environment, the multi-process vs. purely multi-threaded approaches to server program implementation, and the difference between an entirely Wi-Fi connected system of nodes and one where the server computer is wired to the network. Results of the ImagePrep characterization experiments are shown and discussed in Section 5.1.

4.1.2 FAWN Key-Value Data Store Task

Andersen et al. designed, implemented, and characterized the Fast Array of Wimpy Nodes (FAWN) [28, 29], an architecture for a network of low-power, inexpensive nodes of relatively little computing power that can be used for data collection, storage, and retrieval. This architecture was designed specifically to be an inexpensive implementation with low power consumption and to be optimized for solid-state data storage. Andersen et al. advocate for this design because higher-computing-power designs with more expensive higher-performance components are unnecessary for many applications in data centers.

Because retired mobile phones are also an example of inexpensive low-power nodes with limited computing capability and solid-state data storage, the FAWN paper was one of the primary related works that Cluster was compared against. To make a fair comparison between this work and the FAWN paper, a full implementation of the FAWN key-value data store was programmed in the Cluster server program and Cluster node app and characterized on the Cluster hardware.

4.1.2.1 Operation of FAWN Key-Value Data Store

The hardware chosen in Andersen et al.'s original work features 21 PCEngine Alix 3c2 devices, with 500 MHz AMD processor, 256 MB DDR SDRAM (400 MHz), 100 Mbit/s Ethernet, and 4 GB CompactFlash data storage. Each FAWN node consumed 3 W when idle and a maximum of 6 W when deliberately using 100% CPU, network, and flash memory.

The FAWN key-value data store algorithm was implemented in Cluster exactly as described in the original paper. The algorithm generates 160-bit keys for each 50 – 1380 byte data block it needs to store, and uses an in-memory hash table to store 16-bit fragments of the 160-bit keys. For a fetch operation, it looks up the key fragment in memory, attempts to find the relevant block of data in storage from the fragment, retrieves the full key from the returned block, checks for the matching of the full key, and returns the referenced data.

Although the original FAWN work contains details about how the memory is laid out on the flash storage and how the entries in the hash table are interleaved and searched, this part of the implementation was the only detail that was not possible to replicate. This difference was unavoidable because iOS and Android offer developers almost no control over exactly how data is organized in memory. This is common in almost all file systems as well; there is always an interface for storing data with an associated file path and file name, and an interface for retrieving the data by name, but no control over specifically where the data is stored in the large array of flash memory. This discrepancy is an acceptable difference that should not affect the fairness of the comparison in performance between Cluster's implementation and the original FAWN design.

The FAWN data store differs from the common Redundant Array of Independent Disks (RAID) storage method in that it is a much simpler implementation comparable to a large hash table written to flash storage rather than a full file system one might find on a hard drive. RAID storage is another possible application for Cluster that would theoretically be quite well-suited to its architecture. Further study on the Cluster implementation of various level of RAID storage is planned for the future, see Section 6.3.

4.1.2.2 Characterization

The relevant metrics Andersen et al. reported for their FAWN research include read and write speed to the flash memory and energy consumption. The Cluster experiments were designed to measure those metrics as well as network traffic and data size ratios. The statistics recorded for each single run of the FAWN experiments for Cluster are listed below:

1. Timestamp
2. Number of client nodes
3. NetworkReceived: total kilobytes received from the client nodes
4. NetworkSent: total kilobytes sent to the client nodes
5. TaskData: total kilobytes of task-specific data to be processed
6. Number of key-value points to be stored (1600 for the experimental data set)

7. Stage number

8. MainThread loop delay (0 ms)

9. Measured elapsed time to complete the task

The amount of data to be stored was determined based on how much time it takes to complete a single run of the experiment and the amount of memory available on the server computer. Because the four main characterization tasks implemented for Cluster are too functionally different for directly comparing their completion time to each other to be meaningful, the amount of data to be used for each task was adjusted so that each trial of all experiments fell within $\approx 1 - 2$ minutes for 10 connected devices. This constraint allowed repeatable measurement of the task execution time. The number of data points per trial of the FAWN experiment is fixed at 1600. The FAWN characterization task is the only task implemented so far that has multiple distinct stages during its execution: (1) a generate and Store stage, (2) a waiting stage to synchronize all client nodes when they are all finished with stage 1, and (3) a Fetch stage. The Cluster node app is of course not perfect and the FAWN experiment was the last task to be implemented as time was running out to finish this research. Because of this, the FAWN task crashes the Cluster app about 1 in every 20 runs, due to memory leaks or memory access violations that have not yet been able to be eliminated, but significant improvements in the stability of the task were made from the initial implementation that was not stable enough for any meaningful characterization experiments. In order for a trial to be considered valid and be counted towards the collected statistics, all the devices that were connected at the beginning of the run must still be running and connected at the end of the run, i.e. each node must finish the final Fetch portion of the three-stage run.

In Stage 1, the randomly generated 160-bit key and $50 - 1380$ byte data payload are sent together to the client node for storage; 1380 bytes is the maximum Ethernet packet payload size of 1400 bytes minus the key size of 20 bytes. In Stage 2, all nodes that are finished with Stage 1 wait for the others to be ready to move on to Stage 3. The server computer keeps a copy of all keys and the payload data generated for each client so that it can verify the data in Stage 3. In the final stage, all the data (all keys established in Stage 1) is queried in random order. Using the hash table memory lookup of the partial key and the data fetched from the clients, the key and payload are verified and acknowledged by the server.

Because of time constraints on finishing this research, the FAWN task was only implemented for iOS (16 iOS devices were available for testing) and not Android. The Android implementation is planned for future work. In the Cluster node app, there is a persistent database of key-value pairs; the iOS CoreData architecture is similar to how the original FAWN flash storage memory is laid out, fragmented, and queried [88]. The database is stored on the solid-state disk (flash memory) of the iOS phone or tablet. Similar to how the original FAWN architecture retrieves 16-bit key fragments that may match many possible entries in the entire data store and how paging and caching are done for complex file systems on hard drives to improve data latency, blocks of data are retrieved all at once from the slower flash storage, not just the part of the data block that contains the one specific key requested. This block is then cached in memory on the client node so that it doesn't have to be retrieved from the hard drive again the next time any of the keys in that block are asked for. The oldest blocks are discarded when memory is full. The blocks are not a fixed size, they are groupings of key-value pairs whose keys share the same key fragment, i.e. the first 16 bits. While there is a maximum quantity of 1000 key-value data pairs for a block, the CoreData interface makes it possible to only retrieve those keys that exist, not a fixed block of memory that would occupy 1000×1380 bytes = 1.38 MB.

Results of the Cluster FAWN key-value data store characterization experiments are shown and discussed in Section 5.3.

4.2 Cluster Synthetic Tasks

Synthetic tasks were designed not to evaluate Cluster on a fully-implemented application, but instead to test where performance bottlenecks may occur, for certain kinds of tasks, such as on the server or waiting for dependent nodes to pass on the results of their partial calculations to other Cluster nodes.

4.2.1 Server-Prepared Data Task

The ServerPrep task was designed to observe the bottleneck that occurs when a task requires the server to bear an unusually high percentage of the total computation for a distributed task. This task uses thread sleeping and random number generation to achieve this effect. Similar to the ImagePrep and FAWN tasks described earlier, the client nodes are completely independent and do not require data from their neighbors to complete their calculations. Also, similar to the ImagePrep task, the client nodes are hot-swappable in the implementation of the ServerPrep task.

4.2.1.1 Motivation

Normally, the server should only coordinate Cluster node activity and delegate almost all computation work to the client nodes. However, it is possible that acquiring data and pre-processing data on the server machine before it is sent out to the client nodes might take a significant amount of time and processing power. This may occur because the data is being collected in real-time from sensors, retrieved from a hard drive or far-away network resource, or the data is not suitable to be sent to the client node without some initial pre-processing.

4.2.1.2 Operation

The operation of the ServerPrep task is similar to the ImagePrep task, the main differences being the size of the data being passed to the client nodes (and thus the amount of network utilization), and the distribution of work between the server and client nodes. First, the data is prepared on the child processes of the server computer, with each of the n processes spending about 5 seconds in equal parts generating random numbers and sleeping. The two activities are interleaved over the 5 second period, switching between sleeping and generating random numbers between each of 50 generated floating-point data points per client, to simulate wait time for retrieving data and pre-processing the data.

The client nodes are sitting idle during this data generation period. When the data is ready, it is sent to the client nodes as ASCII encoded text in the Cluster packets' payload. When received, the client nodes alter the data with many floating-point multiply, add, and divide operations. This process takes $\sim 1 - 3$ seconds depending on the model of the client mobile device.

The server prepares more data while client nodes are working. The clients send the altered data to the server node and wait for the next set of data to be generated and sent. Each node works asynchronously and independently and does not have to not wait for others to complete their work before they can continue.

4.2.1.3 Characterization

As with the other tasks, the common metrics to be characterized are energy consumption, the trend of completion time vs. number of devices, and the ratio of network traffic to data size. The task runs until 80 data points have been generated, processed, and returned to the server. The statistics recorded for each single run of the ServerPrep experiments are listed below:

1. Timestamp
2. Number of client nodes
3. Number of data points generated for this task (80)
4. NetworkReceived: total kilobytes received from the client nodes
5. NetworkSent: total kilobytes sent to the client nodes
6. TaskData: total kilobytes of task-specific data generated for this run
7. MainThread loop delay (100 ms)
8. Measured elapsed time to complete the task

As this task was designed to do, even with the processor able to devote compute time to the non-idle threads during the sleep operations, and even with multiple processes spread across multiple processor cores, the task of generating numbers quickly becomes a bottleneck for server processor time that cannot be compensated by utilizing multiple server cores or the wait time at the Cluster client nodes. The scope of this research only includes two sets of experiments for the ServerPrep characterization task; one in a typical drywall room, and one in an electromagnetically-shielded room (Faraday cage). Both of these experiments use the same parameters so that the only variable was the wireless environment.

While both of the synthetic tasks described in this chapter have been implemented and confirmed functional in both the single-process C# version of the server program and the multi-process Python version of the server program, time constraints led to only the multi-process implementation being thoroughly characterized. All data collected for the two synthetic tasks and the FAWN data store task are from the multi-process version of the server, which should theoretically perform equal to or better than the single-process version. The results of the ServerPrep characterization task experiments are discussed in Section 5.2.2. Further study is needed and planned to confirm the relative performance of the single-process and multi-process versions of the server program, see Section 6.3.

4.2.2 Dependent Calculations Task

The DependentCalc task was designed to test the performance of a distributed computation task where the Cluster client nodes are dependent on each other to accomplish the work. Many parallel computing tasks operate this way with varying degrees of dependency between the computations associated with the task. The DependentCalc task was designed to simulate that behavior with precise control over how far the calculation can progress before more data is needed from the neighbor nodes and how many neighbors' data are required by each Cluster node.

4.2.2.1 Motivation

Many computationally-intense tasks have been the subject of research for how to parallelize them to speed up the computation, and many such tasks have at least some dependency between the parallel nodes [89, 90, 91]. Some examples of this are Monte Carlo simulations [92, 93] and calculating the digits of pi [94, 95]. Dependency between nodes is impossible to avoid in some computing tasks, so it is important to characterize how a distributed computer architecture based on retired mobile devices performs on these tasks and what bottlenecks appear because of nodes waiting for their neighbors to pass their data.

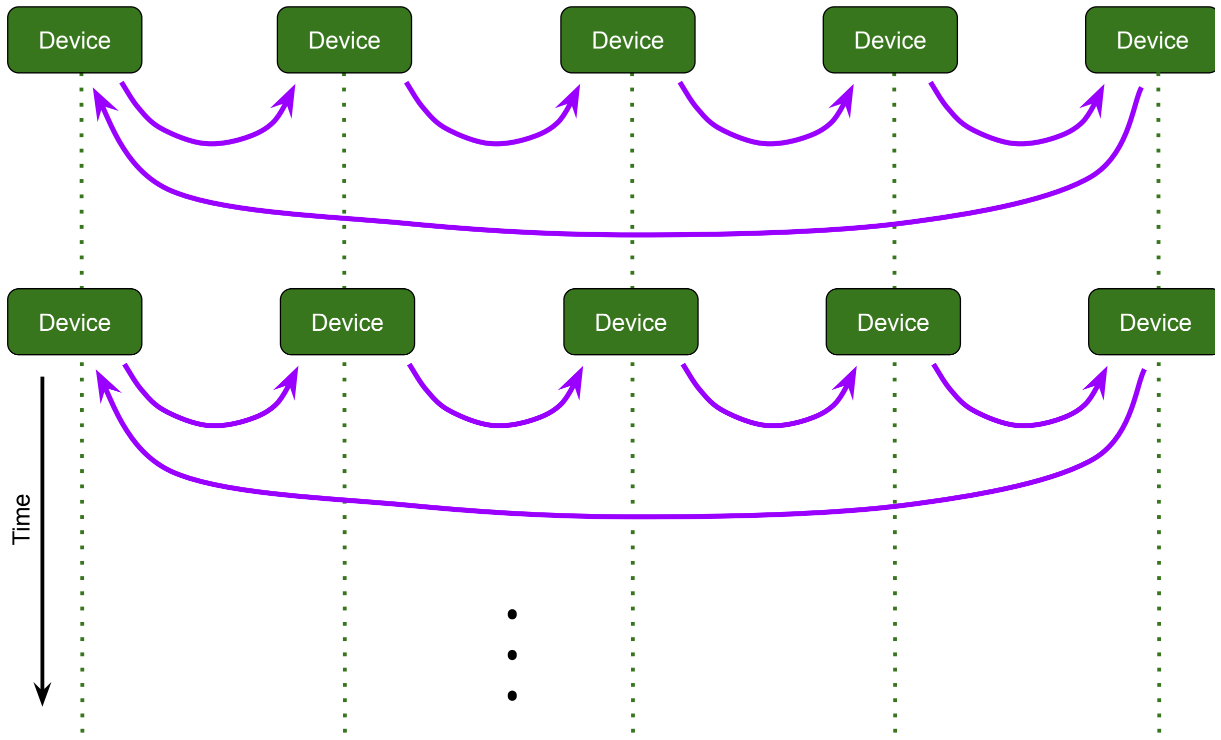


Figure 4.2: Simple Direct Message Passing Between Nodes

In a typical computer network for distributed computers, the nodes are generally arranged in a star or tree formation with a network switch or multiple layers of network hardware in the center and the compute nodes at the leaves of the tree. For computers that use MPI or similar protocols for coordinating parallel tasks, the nodes can simply pass data directly to the nodes that need it without the help of a coordinating server node, as illustrated in Figure 4.2. Cluster certainly could be programmed to do this as well, but it would require an implementation of the MPI library for Swift (iOS) and Kotlin or Java (the languages Android apps are written in) that is supported by the relevant mobile operating systems. While these MPI implementations do exist in Swift [96, 97] and Java [98, 99, 100], they are only supported by Linux and Unix-based operating systems, not for mobile operating systems such as iOS and Android.

Additionally, there have been very few published works on making MPI run on Android mobile devices [101, 102], but even those were not as simple as installing an app on the native operating system. Those implementations required special super-user authority or installation of a Linux operating system on the devices. This approach is contrary to one of the main goals of Cluster, i.e. to be as generically compatible as possible so that any iOS or Android device can run the Cluster app and have the reliability and full features of the native operating system behind it, therefore maximizing the variety and number of retired devices that can be incorporated into Cluster.

4.2.2.2 Operation

The DependentCalc task begins the same as the ServerPrep task, with the server generating 50 random data points to be sent to the client devices. However, this task does not intentionally stress the processor for ~ 2.5 seconds and instead generates the 50 random numbers once, taking less than one millisecond to do so. Just as the other synthetic task, the data is sent encoded as ASCII text and the app performs ~ 1 second of floating-point operations between the checkpoints where a device must receive data from its neighbor.

Figure 4.2 shows the circular path of data dependency in this task. A checkpoint is an event of passing data between nodes so that the computation can proceed. Each node passes its partial calculation results to one neighbor and receives data from a different neighbor. Then the calculation continues for a time until the next checkpoint and this process repeats until the task is finished. The figure does not show how the data must pass through not only the Wi-Fi router, but also the server computer as well.

While it is possible to implement a dependent task for Cluster such that the client nodes can send their data directly to their neighbors through the router and not the server node, that would be a significantly more complex implementation that is outside the scope of this research. That approach would require a partial

MPI implementation to be programmed into the app. That design would also make the hot-swappable feature of the task implementation much more difficult to achieve. While hot-swappability is not critical to most distributed computing applications, the stability concerns of an app-based cluster of retired mobile devices makes this a very useful feature since apps may crash and therefore the device will temporarily leave the cluster until they can restart the app and reconnect to the server. Another effort to undertake in the future is to make the app able to automatically restart when it crashes.

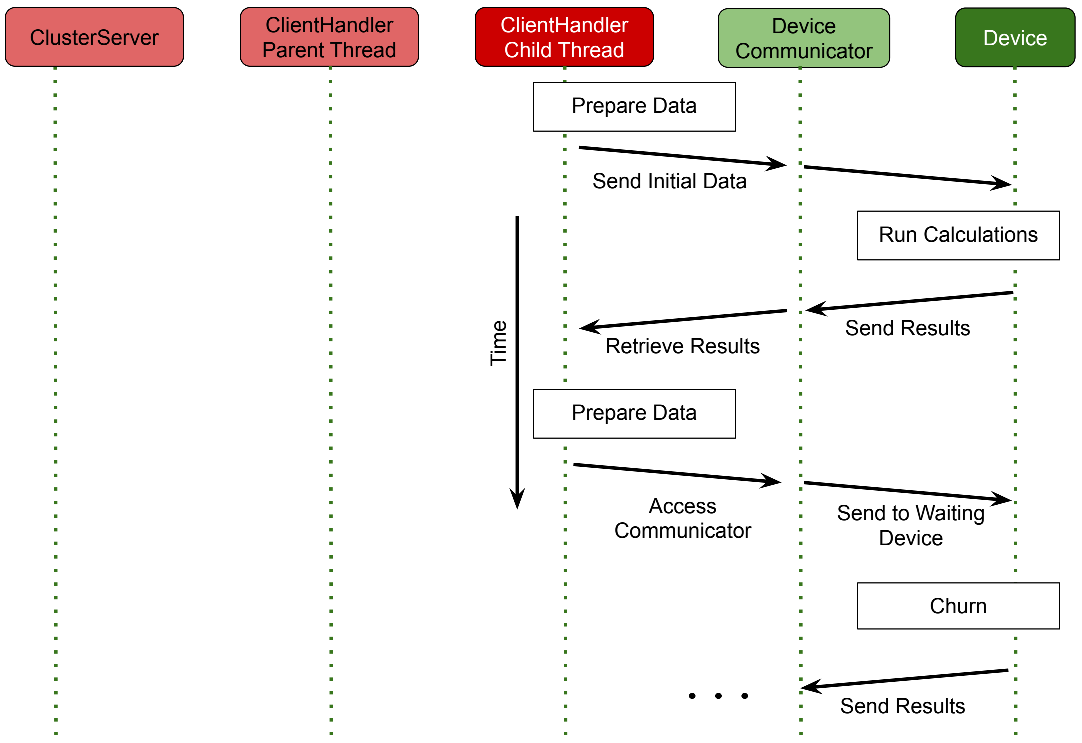


Figure 4.3: Path of Shared Data in a Dependent Task with a Single-Process Server Program

For the single-process multi-threaded C# version of the server program, it is simple to track and coordinate the shared data between nodes. Figure 4.3 shows the path the data takes from device to device through the server program. The data is received by the DeviceCommunicator class, the only thread allowed to receive data from the client node. Then only one level higher in the class hierarchy is needed because the ClientHandler child thread, like all threads within a single process, has access to the same static data structures where the requests for data are stored as they come in. So the ClientHandler can directly send the intermediate data it receives to the node that needs it.

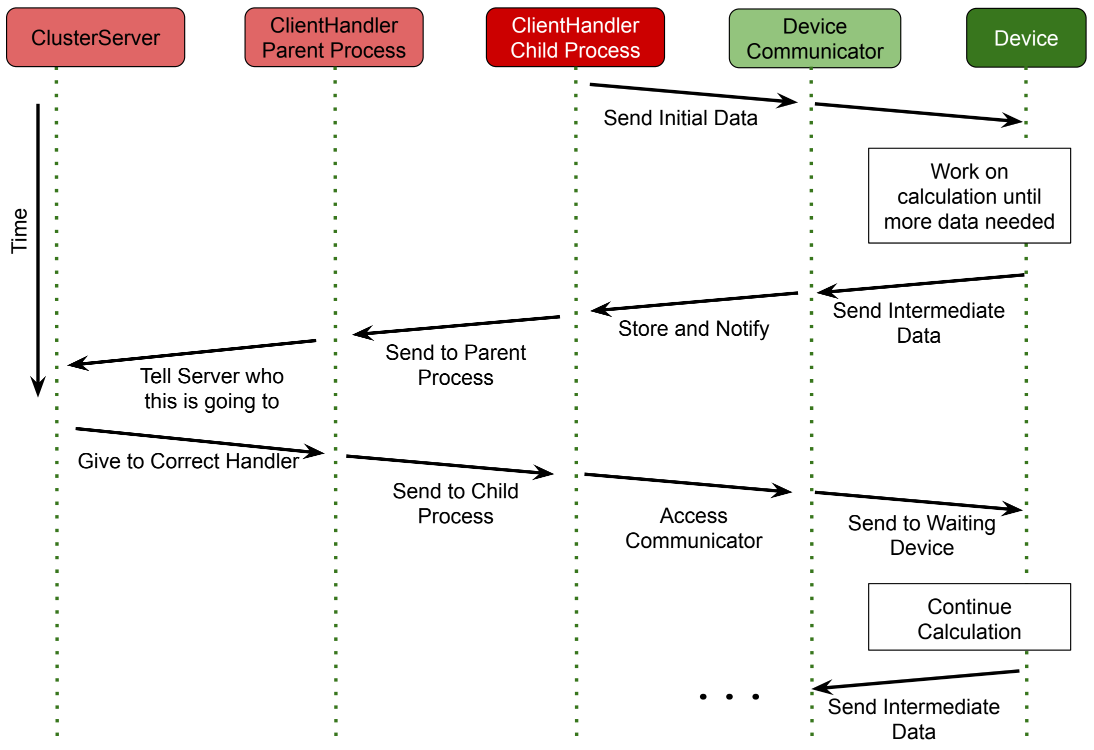


Figure 4.4: Path of Shared Data in a Dependent Task with a Multi-Process Server Program

For the multi-process version of the server, sharing and coordinating between dependent compute nodes is much more difficult as processes do not share memory. The parent process does not have access to the objects that hold the TCP socket connection to the client nodes. Also, in order for the server program to know where to send the data or to access a packet from the process that received it from the client, the data must pass between processes through pipes (communication channels that can pass messages and packaged objects between child and parent processes). Figure 4.4 illustrates that path; it is two layers deeper than the path for the single-process server.

The child processes cannot talk to each other, they can only talk to the parent process. It is also possible to set up pipes between child processes instead of the star network of pipes from the parent to the child processes, but this requires significantly more overhead during the initial connection of the child node and creation of the new process. It also requires exponentially more pipes. The number of pipes for any child process to be able to send messages to any other is equal to the number of edges in a fully-connected graph, $\frac{n(n-1)}{2}$, where n is the number of client nodes plus 1 (for the server node).

Another possible design is to open a pipe from every child to the parent and create a ring of pipes that connect all child processes to each other through one single path. This approach requires some overhead when the processes are created, but much less than the fully-connected topology. The ring configuration works well for tasks where each node only needs to send data to its adjacent neighbor, but would requires more complex coordination for any other task as data would have to travel through each node to get to it's destination. A fixed configuration of any kind except a simple star shape would also be unsuitable for any task where the destination that each node needed to send data to was not constant, but instead changed from checkpoint to checkpoint.

Unlike the other three implemented characterization tasks, the server does not generate more data after the task has begun. All of the work is done by the client nodes. The server is idle except for receiving the intermediate data during checkpoints and passing it on to the correct client node.

4.2.2.3 Characterization

As with the other tasks, the common metrics to be characterized are energy consumption, the trend of task completion time vs. number of devices, and the ratio of network traffic to data size. This synthetic task was designed to have adjustable parameters for how many floating-point operations the Cluster app performs between each checkpoint and how many neighbors it must receive data from in order to proceed. Within the scope of this research, there was only enough time to characterize the task with one fixed set of values for these variables. The experiments were set to have exactly one neighbor's data be required between checkpoints, as illustrated by Figure 4.2.

To determine when to stop the experiment, the task runs until 240 data points have passed through the server node. The statistics recorded for each single trial of the DependentCalc experiments are listed below:

1. Timestamp
2. Number of client nodes
3. Number of data points generated for this run (240)
4. NetworkReceived: total kilobytes received from the client nodes
5. NetworkSent: total kilobytes sent to the client nodes

6. TaskData: total kilobytes of task-specific data generated for this run

7. MainThread loop delay (100 ms)

8. Number of neighbors which a single node depends on for their data (1)

9. Measured elapsed time to complete the task

There are many more experiments that could be run to test the viability of Cluster to run dependent tasks under different parameters, variables, and configurations. The scope of this research only includes two sets of experiments for the DependentCalc task; one in a typical drywall room, and one in an electromagnetically-shielded room (Faraday cage). The results of the DependentCalc characterization task experiments are discussed in Section 5.2.3.

Chapter 5

Cluster Characterization Results

This chapter describes the results of various characterization experiments for each of the implemented tasks described in Chapter 4, both in a regular drywall room environment and inside an electromagnetically-shielded metal-wall Faraday cage. These results are also compared against related works with similar distributed computer architecture implementations.

5.1 Image Preparation Task

Since the ImagePrep task was the first task implemented and is representative of many highly-parallelizable computations, many of the experiments involving changing the design of the server and the network were performed using ImagePrep as a benchmark, with the intention of identifying performance bottlenecks associated with the server computer and Cluster server program. As the number of devices increases in the completion time experiments, the convention for choosing which devices out of all available ones will be participating is that the newest devices are chosen first, presuming that they are the most reliable and have the fastest processors. Therefore the single iPhone XS in the collection was always used when the number of devices connected was equal to 1. Then, once the 1-device experiment was run at least 20 times, the devices were allowed to idle for at least 15 minutes and the second newest device was added, and so on. When an experiment needed n connected devices, the same n devices were always used for consistency. The completion time shown in all figures includes the MainThread loop delay on the server node.

Completion time is the primary performance metric of interest since many cyber-physical system tasks are

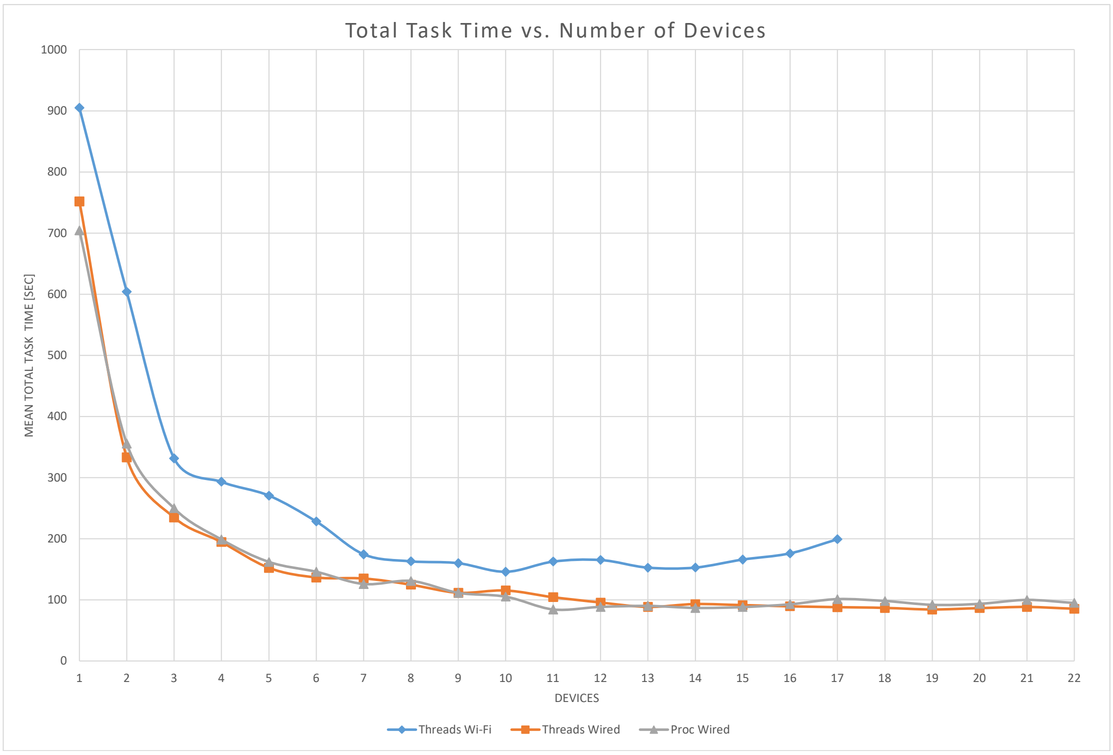


Figure 5.1: Measured ImagePrep Task Mean Completion Time vs. Number of Devices. Threads refers to the single-process version of the Cluster server program while Proc refers to the multi-process version. Wi-Fi and Wired refer to the type of connection between the server and the wireless router.

latency-sensitive and rely on batch processing of periodic uploads of recently-acquired data. For example, in their bioacoustic monitoring research Deichmann et al. [18] programmed 10 audio recorders to capture 1 minute of audio sampled at 44.1 kHz every 10 minutes for 144 recordings per instrument per day. For uncompressed monoaural audio stored in Waveform Audio File Format (WAV), this corresponds to about 5 MB of sound data per recording. Real-time processing of this data would require 50 MB of data to be processed within 600 s, before the acquisition of the next set of recordings.

Figure 5.1 shows the results of three different experiments involving the ImagePrep task. The parameters of the ImagePrep computing task were held constant between experiments: one data set of 2000 images, client node programmed idle time of 100 milliseconds between each image request, the same group of mobile devices, and the same environment and arrangement of devices within the room. The first experiment (shown in blue) uses the first Cluster system design which features all devices, including the server, connected to the network via Wi-Fi and the single-process server program written in C#. The second experiment (shown in orange) is identical but the server computer was wired to the Wi-Fi router with a CAT6 Ethernet cable. The third experiment (shown in gray) features the multi-process server program written in Python, which has an exactly identical algorithmic design, and also a wired server computer.

It is clear from the figure that wiring the server computer to the wireless router eliminated one of the bottlenecks, as changing only that connection reduced the completion time by an average of 54 seconds, regardless of the number of client devices. The graph also shows no discernible difference in completion time between the multi-process and the single-process server programs using the wired server-router connection. This suggests that the bottleneck observed at between approximately 11 and 13 devices, where the completion time settles towards an asymptote for the two wired experiments even when more devices are helping to accomplish the task, was caused by the server program being limited by an underlying hardware constraint as it attempts to coordinate the ImagePrep task across an increasing number of client nodes.

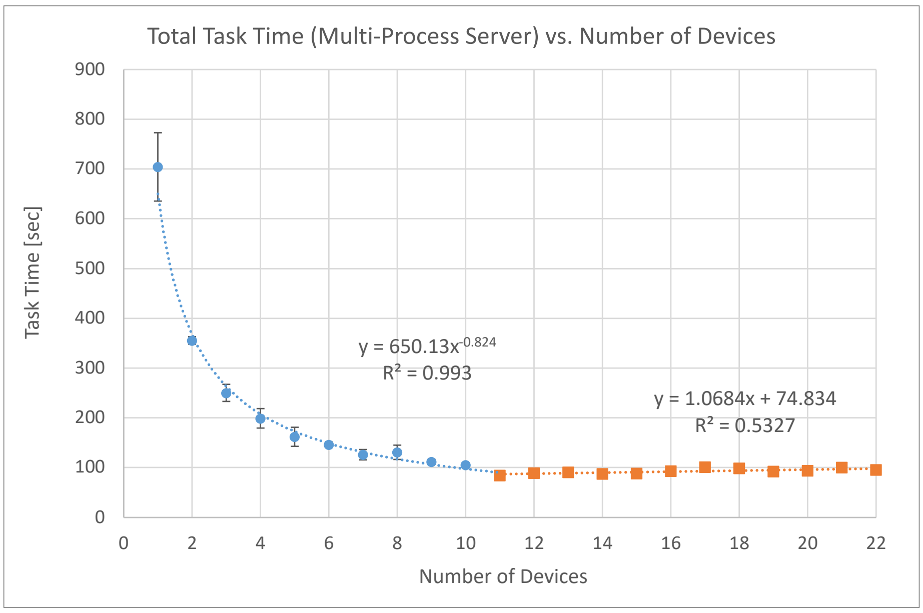


Figure 5.2: Measured ImagePrep Task (Multi-Process Server Program) Mean Completion Time vs. Number of Devices

Figure 5.2 examines the results from only the multi-process server running the ImagePrep task. The error bars shown are one standard deviation from the 20+ runs that represent each data point. The minimum average completion time of 84.2 s occurs for 11 client devices. Two trendlines are shown in the figure, calculated from 1 to 11 devices and then when the trendline becomes more linear from 11 to 22 devices. The

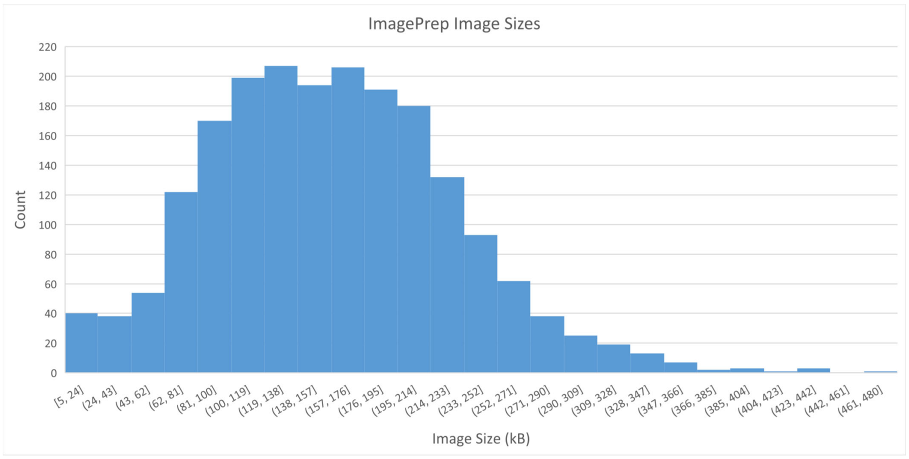


Figure 5.3: Histogram of the file size of all 2000 images in the ImagePrep data set.

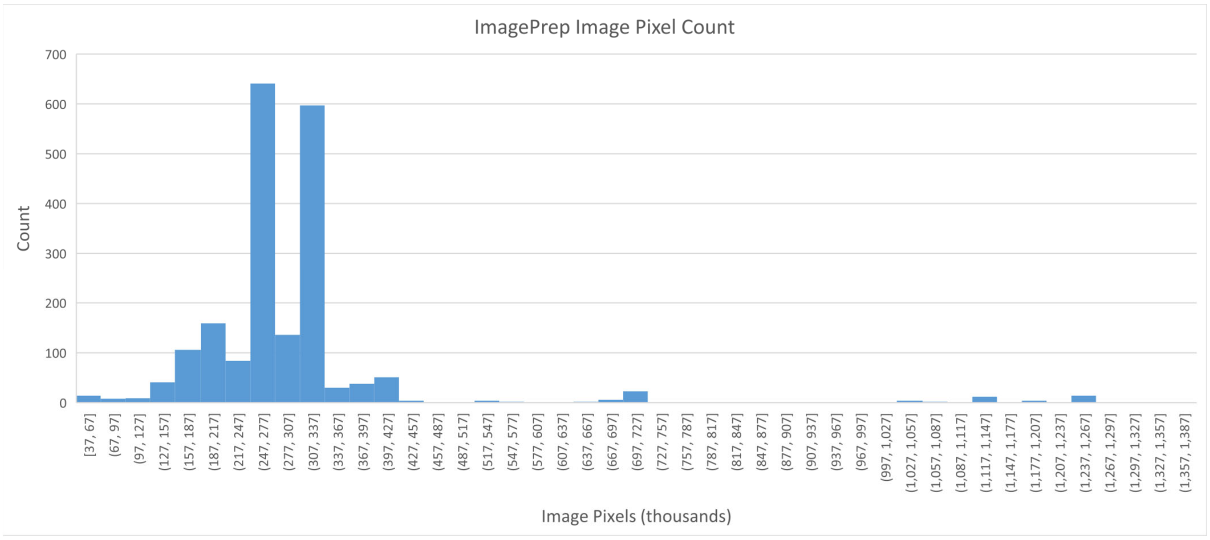


Figure 5.4: Histogram of the number of pixels for each of the 2000 Images in the ImagePrep data set.

right half of this graph shows the saturation of the Wi-Fi router. The older network hardware used for these initial three ImagePrep experiments (NETGEAR N300 EX2700 router, see Section 3.11) was rated for 3.75×10^7 bytes/s and the server computer's network card (capable of supporting 10/100/1000 Ethernet) had a measured effective bandwidth for this experiment of 5.179×10^6 bytes/s on average.

As Figures 5.1 and 5.2 show, the relationship between total task completion time (the time to process all 2000 images) and number of devices is proportional to $n^{-0.8}$ where n is the number of connected client devices, for approximately $n \leq 11$. This is an inversely proportional relationship, roughly $\frac{K}{n}$, where K is the

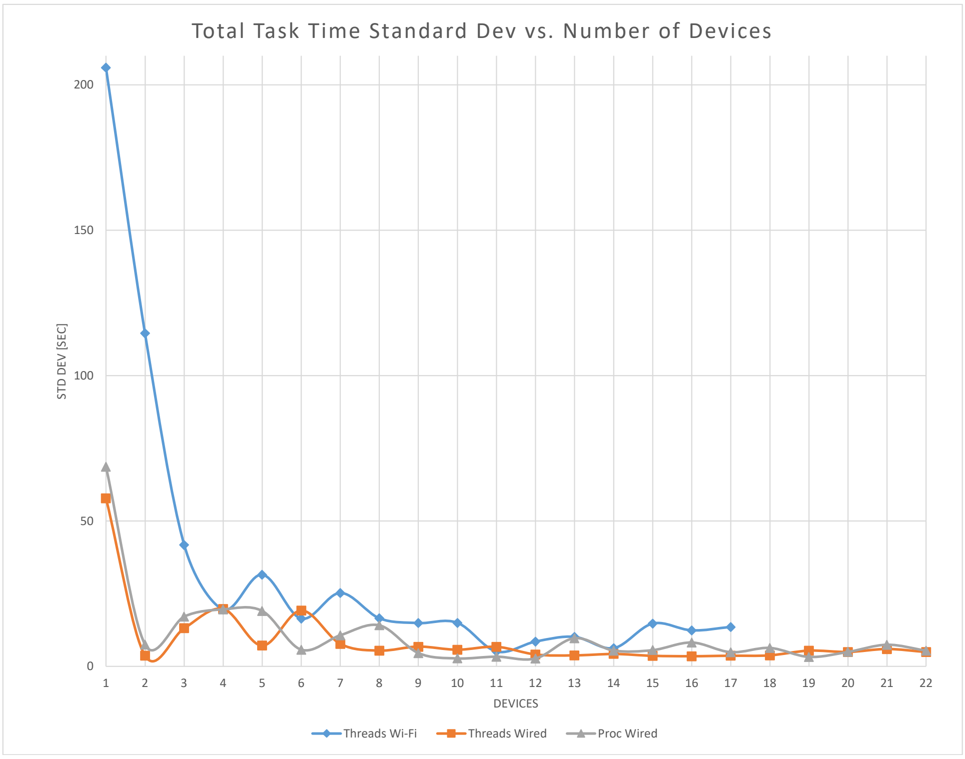


Figure 5.5: Measured ImagePrep Task Standard Deviation of Completion Time vs. Number of Devices. Threads refers to the single-process version of the Cluster server program while Proc refers to the multi-process version. Wi-Fi and Wired refer to the type of connection between the server and the wireless router.

time for one device to process all 2000 images, which is experimentally 704.322 ± 68.7 seconds. The results show that even one reused mobile device node can provide sufficient performance for batch processing of image data every 12 minutes, which should be adequate for many CPS applications. For more than 11 devices, the task completion time appears to increase slowly with additional client nodes (approximately 1.07 s added per each additional device). Since the performance saturates at around 11 client nodes, the results argue for utilizing small clusters of reused mobile devices as fog computing nodes or cloudlets located physically close to sensors and edge devices to minimize data transmission latency.

Figure 5.3 shows the file size distribution of all 2000 images used as the data set for the ImagePrep task. The mean file size is 158 kB with a standard deviation of 69.6 kB. Figure 5.4 shows the dimension distribution of all 2000 images used for the ImagePrep task, expressed as the total number of pixels for each image. The mean number of pixels is 298,800 with a standard deviation of 150,310 pixels. The coefficient of variation $C_V = \sigma/\mu$, where σ is the standard deviation of a distribution and μ is its mean, is a quantitative

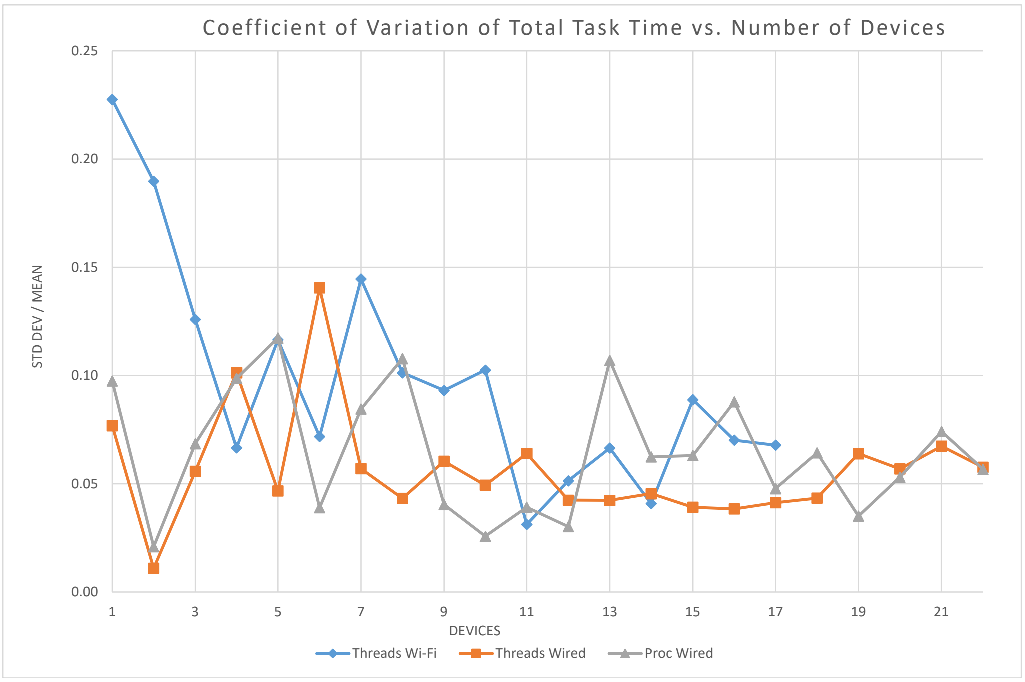


Figure 5.6: Measured ImagePrep Task Completion Time Coefficient of Variation vs. Number of Devices. Threads refers to the single-process version of the Cluster server program while Proc refers to the multi-process version. Wi-Fi and Wired refer to the type of connection between the server and the wireless router.

indicator of the spread (width) of the distribution. $C_V = 0.44$ for the file size and $C_V = 0.50$ for the number of pixels in the ImagePrep data set.

Variation in performance is also an important parameter for cyber-physical systems with real-time or near real-time constraints. Figure 5.5 shows the standard deviation associated with each of the mean task completion time data points shown in Figure 5.1. The absolute standard deviation is much higher when only $1-3$ client devices are connected. Even assuming the standard deviation of each single device's performance is independent of how many other devices are connected, when many devices are connected, the overall performance of the whole cluster has less variation because each node operates independently and asynchronously on the ImagePrep task. Therefore even if one device is experiencing slowdowns from corrupted or missing packets or other misalignments in the synchronization with the server node, the other devices compensate for this and take on more of the work until the task is completed.

The first data point with only one device connected has by far the largest standard deviation because of how long a single device is required to continuously work with minimal idle time and no assistance from other devices. One pattern of behavior observed, but not extensively studied during this research, is that the performance is less predictable (i.e., the variance of the completion time is much higher and the app becomes significantly more likely to crash) the longer the app has been continuously running in a non-idle state. This is a very important characteristic of the Cluster app with significant implications for system reliability and will be studied further in the future, see Section 6.3. Once the number of client nodes reaches four, the absolute standard deviation is consistently below 20 s for both the multithreaded and multiprocess versions of the Cluster server program with a wired server connection to the router. For the multithreaded server program connected wirelessly to the router, eight or more client nodes must be employed in the cluster, indicating that the Wi-Fi traffic between the server computer and router negatively impacts the variation in the task completion latency as well as the total task completion latency.

Figure 5.6 shows the coefficient of variation C_V for each of the data points in Figure 5.1. The observed scaling of completion time and the standard deviation of completion time inversely with the number of client devices is generally consistent with what queueing theory would predict, as is the relatively small and consistent C_V [103]. The distributions of completion time at a given number of client nodes do not follow the same shape as the image size or image pixel count distributions shown in Figure 5.3 and Figure 5.4, respectively, and the coefficient of variation of completion time is about $2\times$ smaller than that of the image distributions. These differences indicate that the image data is not the single determining factor for the variation observed in completion time, and other factors such as network performance are important contributors.

5.2 Task Completion Time Scaling with Number of Devices

As the design of Cluster progressed and characterization tasks were finished being implemented, the network hardware was also updated (Rockspace AC750 router, see Section 3.11). As the implementation of all four main characterization tasks was completed, more experiments were run to compare completion times, data transmission statistics (see Section 5.4), energy consumption (see Section 5.5), and the effects of operating the cluster in an electromagnetically-shielded room (see Section 5.6). Although the ImagePrep, DependentCalc, and ServerPrep tasks have been implemented in both the C# and Python version of the server program, all experimental results other than those shown in Section 5.1 were obtained using the multi-process Python server, which theoretically has equal or better performance than the C# server program.

Figure 5.7 shows the average time to complete each of the four tasks for an increasing number of connected devices. Each data point is the average of at least 20 runs of the experiment. The standard deviation for each of those data points is shown in Figure 5.9 and the coefficient of variation is shown in Figure 5.10. The expected relationship between time and and number of devices is inverse proportionality, $\frac{K}{n}$, where n is the number of devices and K is the amount of time for one retired mobile device to complete the task on its own. The general trend of each of these plots shows the expected inverse relationship. The trend is more clearly seen in the log-log plot of Figure 5.8. Three of the tasks, DependentCalc, FAWN Store, and FAWN Fetch, exhibit near-ideal scaling with the number of client nodes. The trend appears independent of the node heterogeneity, where the oldest devices are added last to the Cluster client node pool, indicating that the performance of even ten-year-old retired mobile devices may not limit overall Cluster performance on some tasks. The ImagePrep and ServerPrep tasks exhibit limited scaling at approximately 18 nodes and 10 nodes, respectively. Reasons for this will be discussed below. The results for the FAWN data store task will be discussed in Section 5.3.

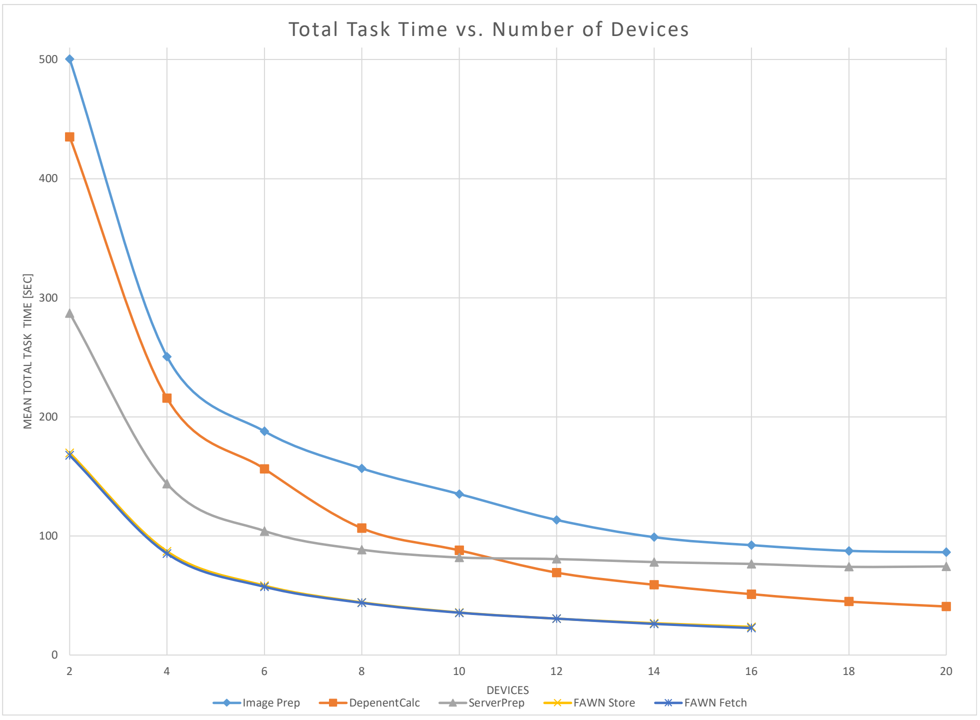


Figure 5.7: Measured Completion Time vs. Number of Devices for All Tasks

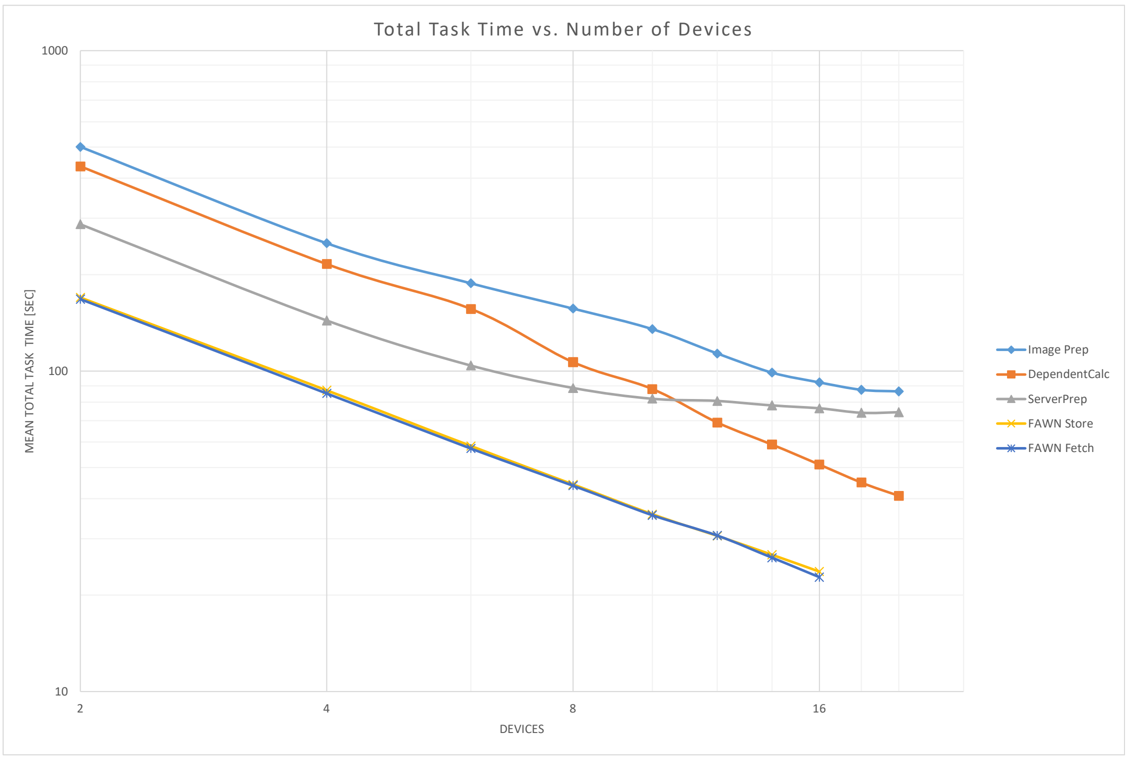


Figure 5.8: Measured Mean Completion Time vs. Number of Devices for All Tasks (Log-Log Scale)

5.2.1 ImagePrep Task

For this set of experiments, the Wi-Fi router used (Rockspace, Section 3.11) was also rated for 3.75×10^7 bytes/s and the server computer's network card, which has sockets open to all connected devices, was communicating at a measured maximum rate of 5.471×10^6 bytes/s on average during the most network-intensive task, ImagePrep. This means that the router was not saturated during the ImagePrep task or any other task. For the ImagePrep task, the amount of time spent transmitting the images to and from the server is significantly more than the amount of time spent on computation (altering the images), specifically 2 to 10 times larger depending on the size of the image. Thus the bottleneck most likely is with the network card on the server computer. Twenty or more devices communicating with one server computer that has a limited number of multicore processors is likely to encounter a communication bottleneck at some point as the network is scaled up, because the Cluster server program is servicing requests for each client device. A high-performance server with sufficient parallelism to support enough client nodes to saturate the Wi-Fi router would likely be better used to compute the application tasks directly on its own. Therefore, for network-intensive tasks, a different Cluster design would be required to move past this asymptote, such as using multiple low-performance server computers to coordinate the efforts of the client nodes.

Note that any comparison of completion time between different tasks has no meaning since the tasks have many parameters that could be adjusted that would affect the total time for finishing the task, i.e. number of data points, amount of client idle time, size of the data set, and more. Additionally the tasks are each quite different, with too many variables to make comparing their completion time performance meaningful in any way. However, some metrics do have meaningful comparisons between tasks, such as the point where asymptotes occur as the number of client nodes increases, energy consumption, and network statistics.

Another thing to note about the data in Figures 5.9 – 5.10 is that the ImagePrep task is the only task to have non-constant standard deviation for an increasing number of connected devices and has also consistently the largest standard deviation. That task also has the highest coefficient of variation of any task. This is most likely because it is difficult to synchronize the sending of a large file in fragments from one device to another and to have it reassembled in the right order at the receiving node. TCP has a well-established protocol for doing this for file downloads, but socket libraries for user-level programming don't have functions for sending entire files with confirmation, just simple send and receive functions, and these were used to implement the lightweight file transfer protocol in Cluster. A queueing theory analysis of these results is beyond the scope of the present work, but is suggested as a topic for future research.

5.2.2 Server-Prepared Data Task

The ServerPrep task was designed to induce a computation bottleneck at the server node. This behavior can be observed in Figures 5.7 and 5.8 that show the ServerPrep task becoming approximately a constant time with 10 or more devices connected. The ImagePrep task is the only other task observed to have an asymptote of completion time for under 20 connected devices, shown in Figures 5.1, 5.2, 5.7, and 5.8. Since the ServerPrep task performance stops scaling at fewer devices than the ImagePrep task, the network performance does not limit ServerPrep task performance. As described in Section 4.2.1.2, the ratio of server pre-processing time per client node to client node execution time ranges from about 1.67 to 5 for this synthetic task, depending on the client node. Depending on the hardware chosen for the server, an application with this type of skewed workload characteristic would not fully utilize a cluster with more than a few client devices.

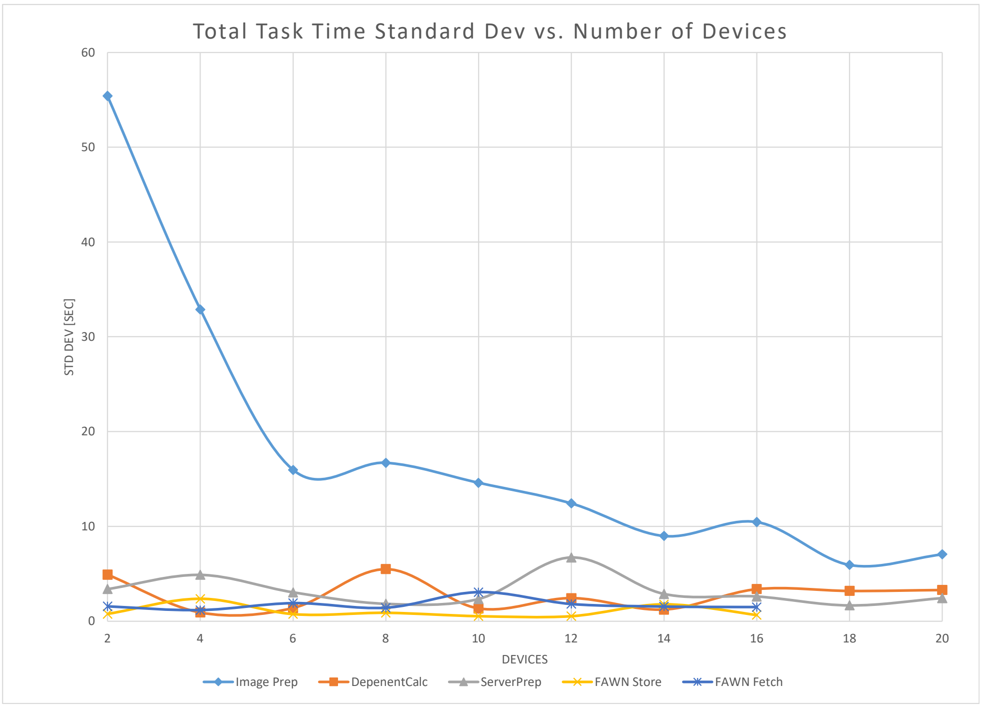


Figure 5.9: Measured Completion Time Standard Deviation vs. Number of Devices for All Tasks

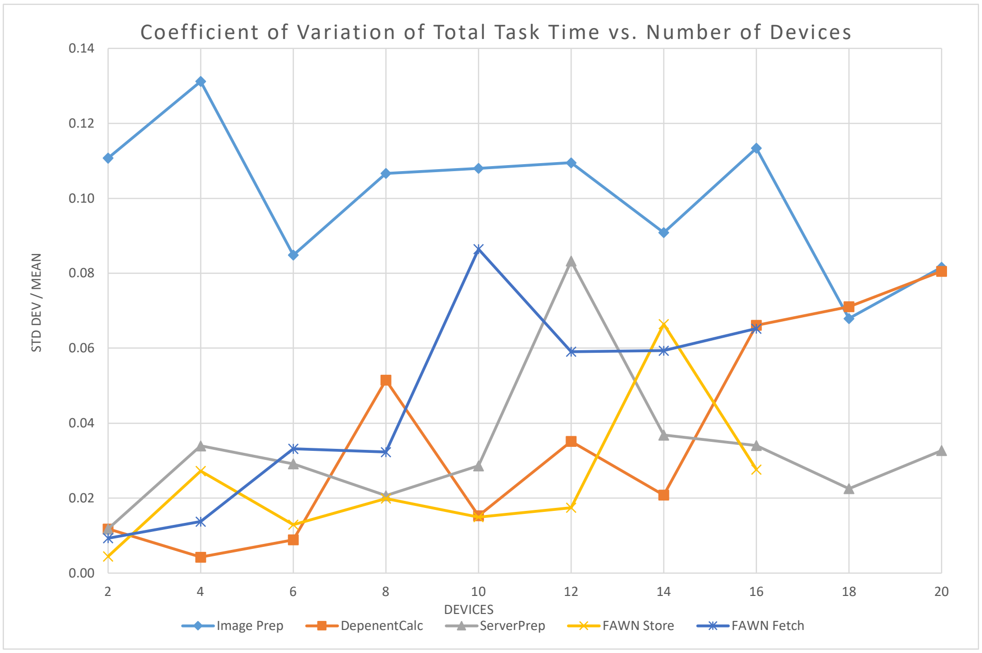


Figure 5.10: Measured Coefficient of Variation of Completion Time vs. Number of Devices for All Tasks

5.2.3 Dependent Calculations Task

Figure 5.8 shows that the DependentCalc synthetic task can continue scaling beyond 20 client nodes, despite the potential bottleneck created at the server, which is responsible for passing messages between the client nodes. The synchronization barriers occur at approximately 1 second intervals as the arbitrary computation included in the version of the DependentCalc task studied in this work does not exercise any significant differences in the computational throughput of the client nodes. In an implementation with a larger age distribution of reused devices or a workload that requires significant computation time, the oldest devices with the least performance are likely to limit the overall performance of tasks with data dependency. Further study of this is suggested as future work.

5.3 FAWN Task

As shown in Figures 5.7 – 5.10, the FAWN task was only characterized up to 16 devices because of time constraints on the implementation. The FAWN implementation needed to be precise and so to maximize available time, the FAWN task was only implemented on the iOS Cluster app, not the Android Cluster app. Thus the data for the FAWN task is all from a Cluster built using iPhones and iPads. Further study of Cluster as a FAWN implementation may be needed in the future, and if so, the Android app will feature this task. It is possible that an Android implementation may perform better than the iOS realization as Android has a completely different interface for accessing the non-volatile flash storage on the mobile devices.

The FAWN task was characterized in two stages, Store and Fetch, to be able to compare the read and write speeds to the original work. The Fetch portion of the task attempts to fetch all key-value pairs that were written to the app's database in the Store stage in random order. The execution time of both stages can be observed to be identical from Figures 5.7 and 5.8.

5.3.1 Comparison to the Original FAWN Work

In their paper [28, 29], Andersen et al. achieved a random read rate of 1424 queries per second (qps) and a write rate of 110 qps for a 3.5 GB dataset. Their reported speeds were significantly faster with a comparable size dataset to the one tested with Cluster (250MB), 6824 qps for random reads. Random writes were not reported for this dataset size.

The Cluster implementation achieved FAWN data store rates, for 1 kB entries, of 67.65 ± 1.76 qps (52.06 kB/s) for random reads and 74.14 ± 15.14 qps (52.06 kB/s) for random writes. This seems significantly slower than the original FAWN work, but it is important to note that the Cluster implementation has the additional overhead of having to send the data over a network before it can be written to the database. Similarly for the fetch operation, the data must be sent to the server node after it's retrieved from the database. FAWN originally targeted data center applications; the performance achieved by Cluster is adequate for cyber-physical systems applications that likely involve relatively small data sets and a limited number of simultaneous queries.

This overhead certainly constitutes more than half of the total time consumed during the characterization task. Therefore the rate reported here is the rate at which data could be stored in real time from the perspective of the server if it sent the data as fast as possible to the connected devices, which is not the same metric as the original FAWN paper reported. Additionally, this was the fastest store and fetch rate with 16 client nodes and no asymptote of task completion time was observed for the FAWN implementation up to 16 devices, so the maximum rate is likely even faster than what could be measured in these experiments.

A new experiment is needed to characterize how fast the data can be added to the database or fetched from the database without interacting with the server node. This experiment would not interleave the server interaction with the reads and writes and would do all database interactions all at once and measure that time separately from getting or sending data to the server before or after the bulk read and writes. This experiment is planned for the future, see Section 6.3.

Andersen et al. also reported the rate in MB/s of sequential reads and writes. This statistic is currently not possible to measure with Cluster's implementation of FAWN because the iOS CoreData interface doesn't provide an interface for reading all of the stored data in sequential order. However, there is an interface for getting all data in a particular table of the database which may provide the fastest possible read speed comparable to sequential reads. That characterization and a similar one for maximum write speed are planned for the future.

5.4 Network Statistics

Another important metric for Cluster tasks is the amount of data required to be sent across the network throughout the execution of tasks. Table 5.1 shows how much total data each task requires to be passed between client and server.

Task	Sent to Clients	Received from Clients	Task Data
ImagePrep	319458.266 ± 3100 kB	146651.266 ± 8500 kB	316173.561 ± 0.0 kB
FAWN Store	1219.770 ± 13.702 kB	0.0 ± 0.0 kB	1203.008 ± 13.664 kB
FAWN Fetch	545.063 ± 124.096 kB	1023.630 ± 193.976 kB	1203.008 ± 13.664 kB
ServerPrep	240.586 ± 37.346 kB	241.464 ± 36.601 kB	238.581 ± 36.220 kB
Dependent Calculation	$173.41 \pm 78.19 \ \frac{\text{kB}}{\text{device}}$	$154.59 \pm 63.25 \ \frac{\text{kB}}{\text{device}}$	$0.974 \pm 0.124 \ \frac{\text{kB}}{\text{device}}$

Table 5.1: Average Network Data Traffic per Task

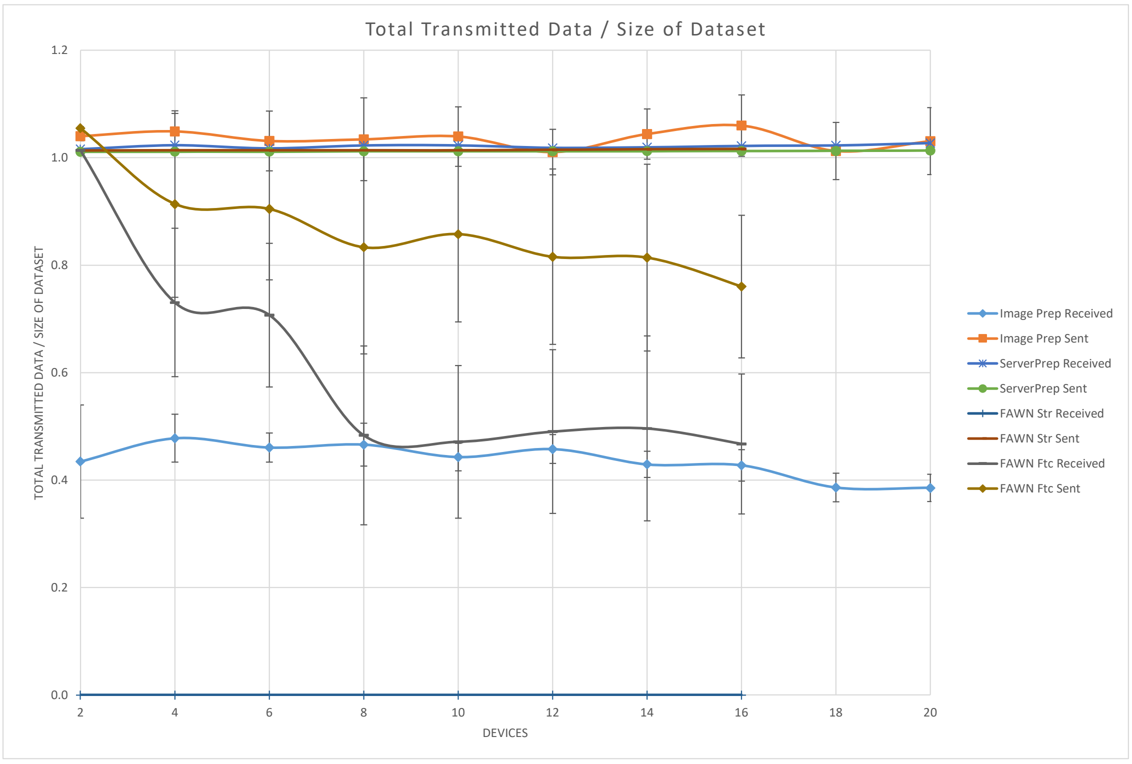


Figure 5.11: Transmitted Data / Total Task Data vs. Number of Devices

The data in Table 5.1 is constant for any number of connected devices, except for the DependentCalc task which generates unique data only at the start of the task and never again, and the amount of generated data is proportional to how many devices there are. The tasks implemented to characterize Cluster were designed to have a fixed amount of data that needs to be processed to complete the task. For most tasks, this results in a constant amount of network traffic regardless of how many devices are connected. This can be observed in Figure 5.11.

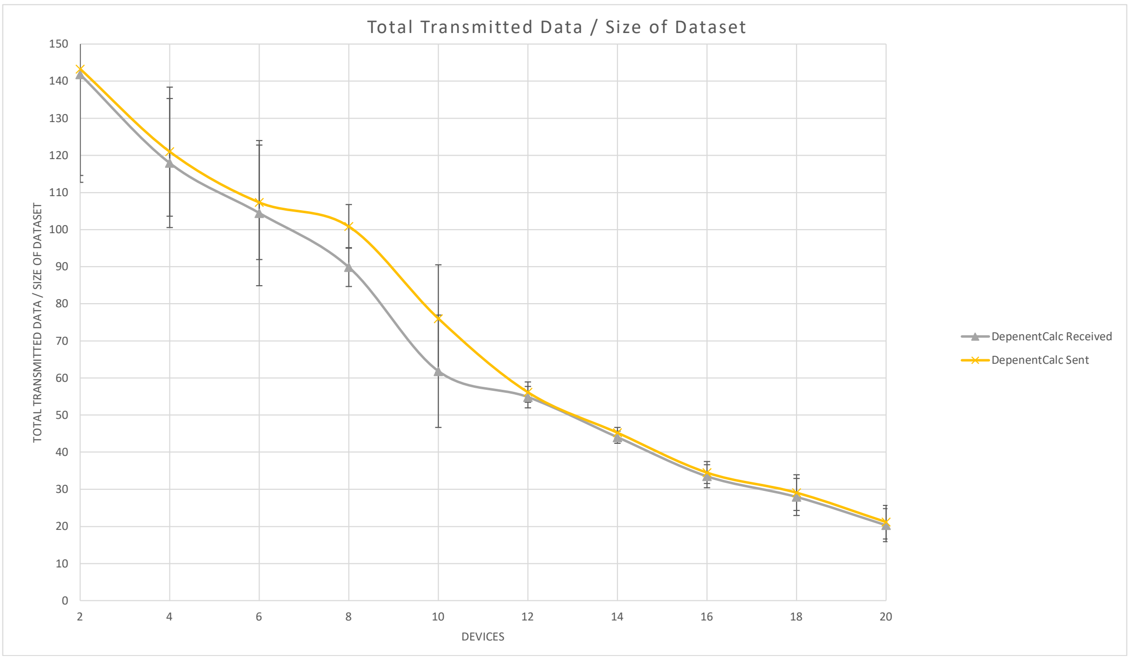


Figure 5.12: Transmitted Data / Total Data for DependentCalc Task vs. Number of Devices

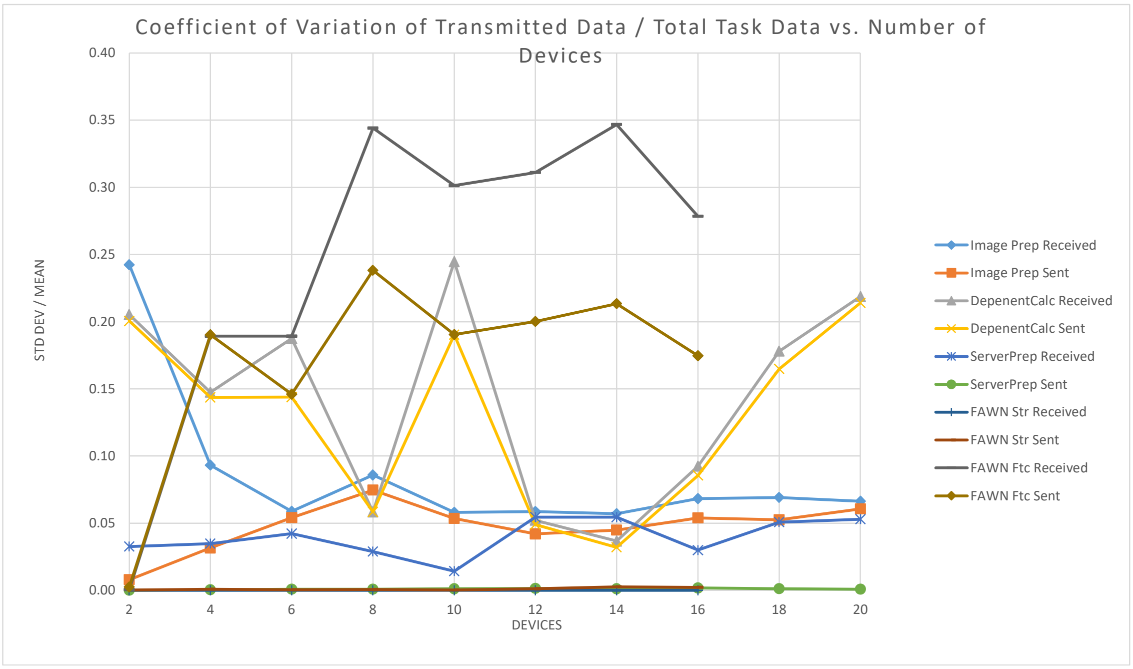


Figure 5.13: Coefficient of Variation of Transmitted Data / Total Task Data vs. Number of Devices

In terms of network activity, another metric than can be measured from knowing how much data is sent across the network and the size of the initial dataset is the ratio of data transmitted divided by dataset size. This ratio, called the ratio of network data propagation for the purposes of discussion, describes how much network traffic is needed for a particular task compared to how much data there is to process. This can be an important metric to measure for Cluster and similar distributed computer architectures because it may help determine how suitable a task is for Cluster, which uses Wi-Fi connectivity and retired devices compared to more powerful compute nodes with gigabit wired connectivity.

Tasks that require large amounts of network traffic will run slowly and be less efficient on Cluster than on traditional distributed computers because of the limitations of Wi-Fi and a single coordinating server node through which all data passes. The lack of an asymptote in completion time for the DependentCalc and the FAWN tasks for the experiments presented here indicate those types of tasks are more scalable on Cluster than other task types and are therefore the most appropriate types for this kind of architecture to be able to run efficiently.

Note there are four overlapping lines in Figure 5.11 that hover just above a network data propagation ratio of 1.0. This indicates that for ServerPrep (both sending and receiving), FAWN (sending), and ImagePrep (sending), the amount of data required to be sent across the network is almost equal to the amount of data there is to process. The small amount that the ratio exceeds 1.0 is because of packets that needed to be re-transmitted and the extra packets that do not contain data, but rather commands required to coordinate the task execution.

The total amount of network traffic is still double the size of the dataset in cases where the ratio is 1.0 because the data needs to be sent to the client nodes and then sent back. In some cases, the amount of data received at the server is smaller than the total that was sent to the client nodes. This is the case for FAWN (data received during the Store stage is 0), ImagePrep (image files come back smaller after processing), and erroneously for both sent and received metrics for the FAWN Fetch stage. The way the network traffic is measured is, at the server node, every byte sent and received from each open socket is added to a global accumulating sum. The reason the FAWN Fetch statistics have such a high standard deviation in Figure 5.11 is that the FAWN task is the least stable of all implemented tasks due to memory access bugs and errors that were not able to be fixed in time for these experiments. Therefore, in almost every run of the experiments (less so with only 2 or 4 connected devices), some devices would experience app crashes between the first and third stages (Store and Fetch with a synchronizing stage in between) of the FAWN task characterization experiment. This app failure resulted in the data being unavailable for those devices during those runs, which meant the total amount of data transmitted across the network was less than the total size of the dataset for all devices. Theoretically, the ratio should be 1.0 for the FAWN Fetch stage as well if there are no app crashes. This can be seen from the data points for 2 connected devices where the app rarely crashed.

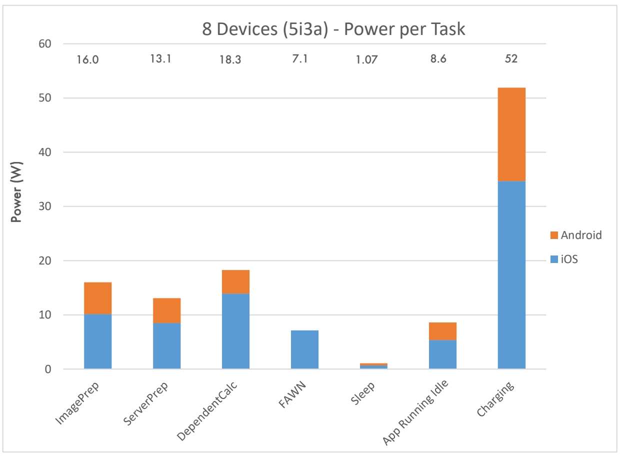
The ratio for DependentCalc is beyond the y-axis scale of Figure 5.11, so that data is shown in Figure 5.12. The ratio is very high compared to other tasks because a small amount of data (~ 1 kB per device) is generated at the start of the task and no new data is generated for the rest of the task's operation, but data is still continuously passed between client nodes through the task execution. This is similar to how many parallel scientific computing applications like calculation of the digits of pi would work: an initial seed of data would be given to each device and then the calculation would proceed at the client without needing any more information from the server, each client only needing intermediate results from neighboring client nodes to proceed.

Ignoring the data for the FAWN Fetch task due to Cluster app instability during its characterization, as discussed above, Figure 5.13 shows that for most tasks, the total transmitted data to total task data has a narrow distribution, with $C_V < 0.1$ for all tasks except DependentCalc once the number of client nodes is four or more. The distribution (variation) for DependentCalc is generally higher but still relatively narrow ($C_V < 0.25$). For the tasks studied in the present work, the main conclusion is that network traffic does not vary widely as the number of client nodes increases.

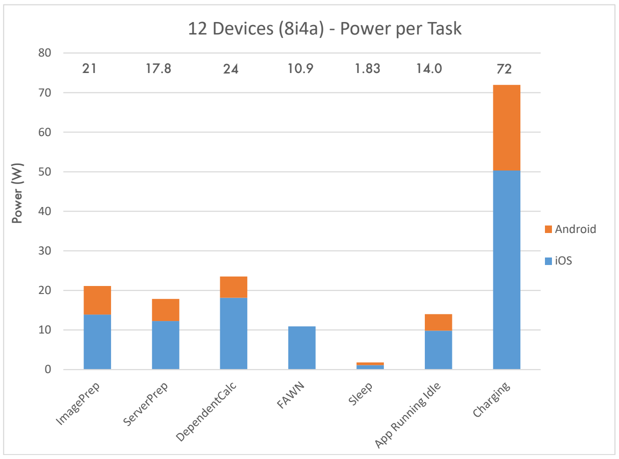
5.5 Power

Energy consumption is a critical metric for any distributed computer. Data centers and supercomputers consume $100 - 200$ times more energy than a typical office building [104, 105]. The majority of that electricity cost is the cooling system (about 50%) with only about 36% of total energy consumed by the computers and network hardware [106, 107].

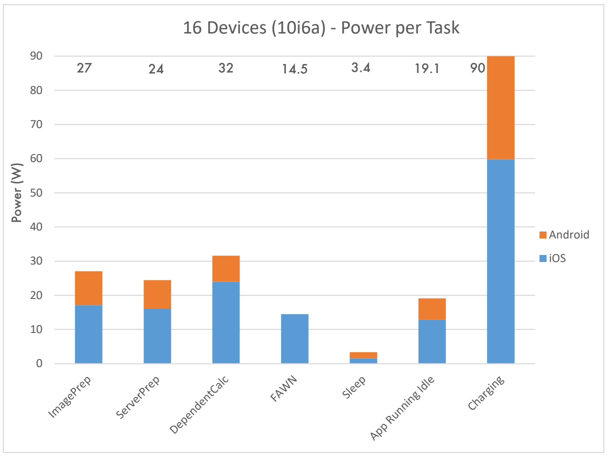
To measure Cluster's energy consumption, a Poniie PN2000 Electricity Monitor was used to measure the total energy consumed by performing many runs of each task individually over several hours until the energy consumption reported by the meter constituted at least 2 significant figures. The client nodes each have a



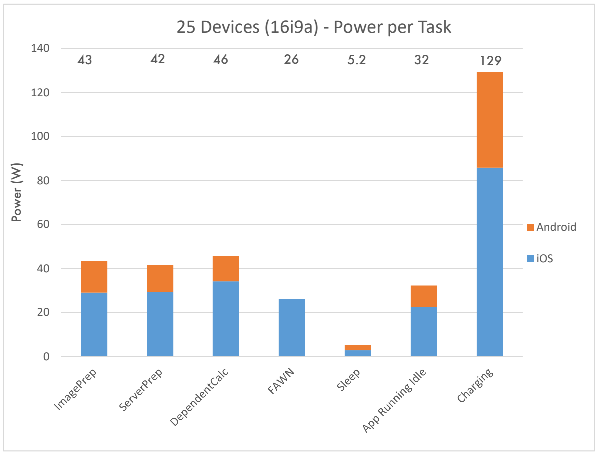
(a)



(b)



(c)



(d)

Figure 5.14: Measured Power per Task vs. Number of devices. The number of Android devices is denoted by "a" and the number of iOS devices is denoted by "i".

battery, but Cluster was designed for all the mobile devices to remain powered by the charging cable at all times. Therefore the power measurements and all other experiments were performed with the batteries 100% charged so that all electricity consumed would be from the wall outlets. Devices were plugged into AILKIN 4-Port USB Brick adapters that each had four USB charging ports. The results of the power measurement experiments are shown in Figure 5.14. The data is also represented as linear plots in Figures 5.15, 5.16, and 5.18, with different operating systems on separate graphs. The title of each chart indicates how many Android devices and how many iOS devices ("a" corresponds to Android, "i" corresponds to iOS) were used for each of those data points, with colors also indicating the proportion of the

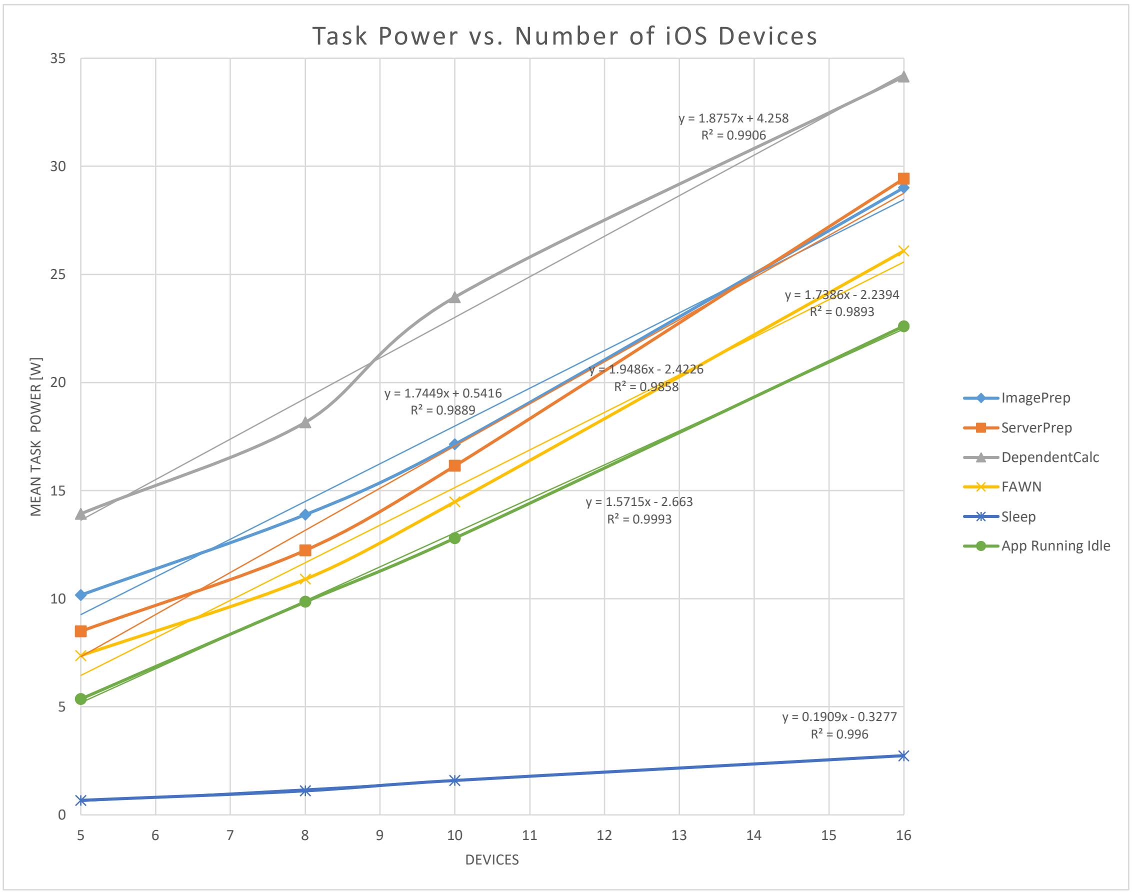


Figure 5.15: Measured Power vs. Number of iOS Devices

total power per operating system.

The original FAWN implementation [28, 29] consumed 3.9 watts per node when idle, 4.3 W/node on storing data, and 4.7 W/node on fetching data. Cluster consumes 1.29 W/node when the app is running and idle, and 1.63 W/node when running any stage of the FAWN task. The nodes use even less power when the device is asleep (while still plugged into wall power); only 0.21 W/node. The other tasks use only slightly more energy then the FAWN task, a maximum of 1.88 W/node for the DependentCalc task, which dissipates 45.8 W when all 25 devices are running that task.

Another system of used mobile devices, created and benchmarked by Switzer at al. [26] constructed an architecture similar to Cluster out of Google Nexus 4 and Nexus 5 devices running Ubuntu Touch instead of the native Android operating system; further discussion of this work appears in Section 5.8.2. Their system consumed 0.9 W/node when idle and 2.8 W/node when actively computing. In both cases, Cluster operates with lower energy per node when actively computing and when idle. This is likely due to the low-power design of mobile devices and the native operating system being more optimized for energy efficiency than a third-party Linux OS.

Figure 5.17 shows the average power per device for iOS and Android for each task. Here it can be seen that the iOS devices tested consume more power than Android devices in almost all instances except for sleep mode, and it is unknown for the FAWN task since the FAWN task was not yet implemented for Android.

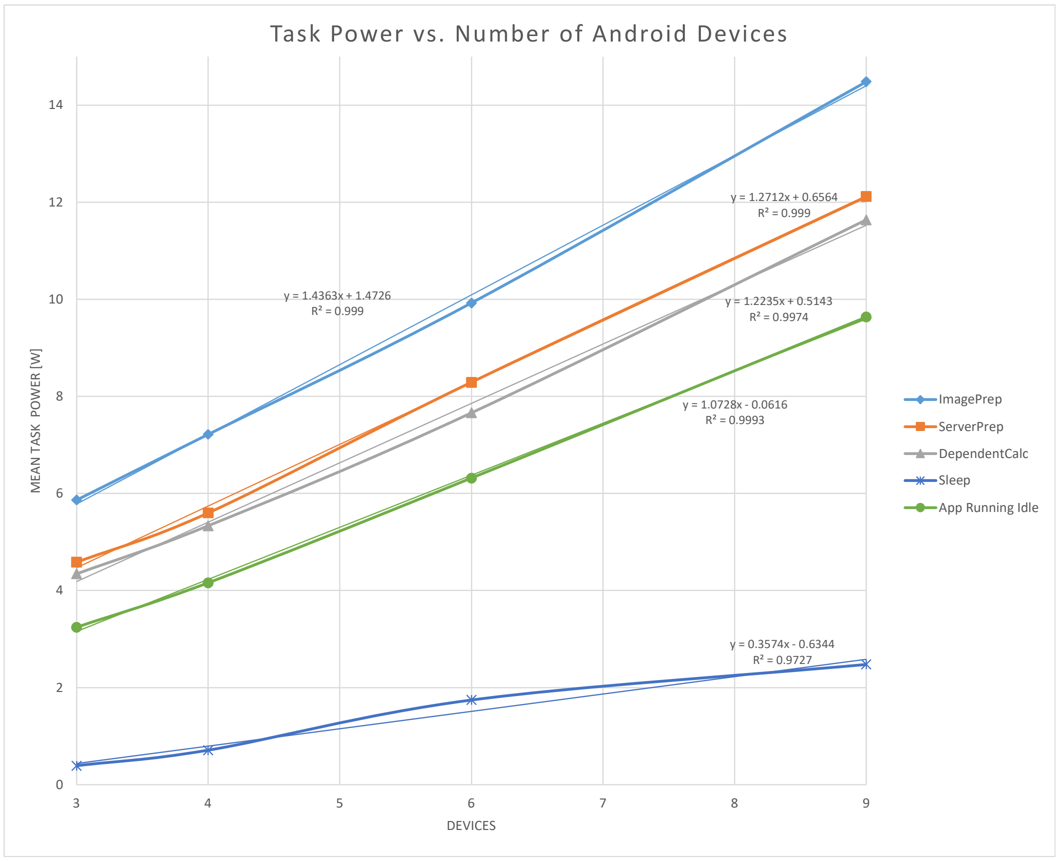


Figure 5.16: Measured Power vs. Number of Android Devices

5.5.1 Computational Carbon Intensity

Switzer et al. [26] established the metric Computational Carbon Intensity (CCI) to evaluate the CO_2-equivalent (CO_2e) released per unit of computation work across the entire lifespan of the compute nodes in a distributed computer made up of used mobile devices and compare it to alternate

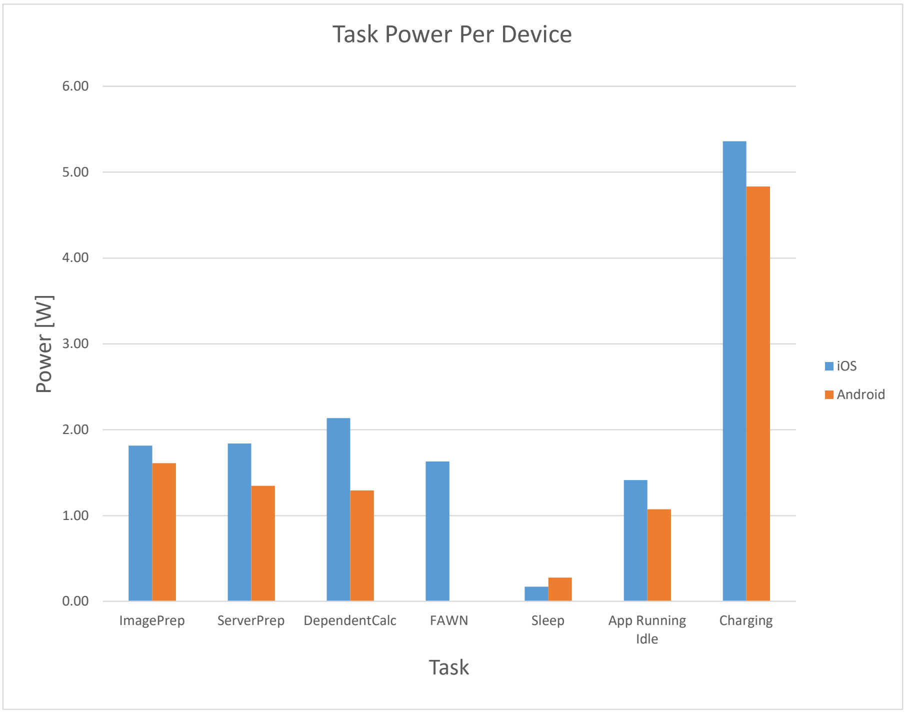


Figure 5.17: Measured Task Power Per Device

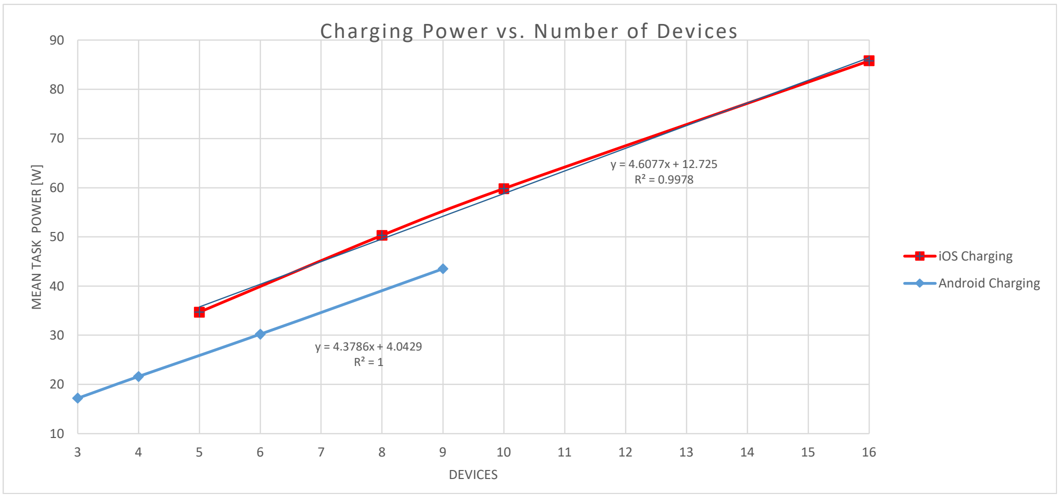


Figure 5.18: Measured Charging Power vs. Number of Devices

implementations using conventional data center servers. The equation for CCI proposed by Switzer et al. is:

$$\mathrm{CCI} = \frac{\sum_{\text{lifetime}} \mathrm{CO_{2e}}}{\sum_{\text{lifetime}} \text{ops}} = \frac{\mathbb{C}_M + \mathbb{C}_C + \mathbb{C}_N}{\sum_{\text{lifetime}} \text{ops}} \tag{5.1}$$

The quantity ops in this case is the number of Operations Per Second, which varies depending on the task, but is generally equal or proportional to either (1) flops, floating-point operations per second, a common measurement of processor work, or (2) bytes per second, a measure of how much data is being processed for a task and how quickly.

$\mathbb{C}_M$ is the carbon emissions from manufacturing the device originally. It is a one-time cost and is assumed to be zero for the purposes of this work because the devices are reused and thus already manufactured. Switzer at al. also assumed this value to be 0 for their implementation with additional carbon cost associated with replacing the devices' batteries about every 2 years. The Cluster implementation does not require batteries to be replaced. The devices may have defective batteries and it makes no difference since the devices are wired to power at all times. However, as Switzer et al. state, batteries can provide a backup feature by serving as an uninterruptible power supply (UPS) that may be important for a cyber-physical system.

$\mathbb{C}_C$ is the carbon associated with the energy of compute. This is the carbon footprint of the energy generated and transmitted on the power grid. This metric varies depending on how the energy was generated and may even be 0 if the energy is clean renewable energy. Similarly, $\mathbb{C}_N$ is the carbon associated with the energy consumed by the networking hardware, including Wi-Fi routers and the Wi-Fi antennas in the mobile devices and the server computer's network card. The power consumption of networking and compute for Cluster were measured together since the energy of the Wi-Fi hardware on the mobile devices could not be measured separately in these experiments.

Below is presented the calculation of CCI for two different scenarios of Cluster running the ImagePrep task in order to compare the carbon intensity against the system presented by Switzer et al. Because the performance of the ImagePrep task saturates at 18 devices, the two scenarios will assume 18 devices running the ImagePrep task at 2 different duty cycles for a period of one month (31 days). The scenario corresponds to a limited-duration cyber-physical system task such as an image-based biodiversity or environmental monitoring study. For example, the study in Deichmann et al. [18] consisted of a 14 day first study period followed by a second 17 day period.

Scenario 1 consists of a very small compute task duty cycle of one run of the ImagePrep task per day, which at 2000 images processed per run yields a total of 62,000 images processed in the 31-day lifetime of the study. This scenario corresponds to a daily upload of images captured by remote sensing nodes. Since the Switzer et al. paper presents one of their studies in terms of mgCO_2-e/Mpixel (milligrams of carbon dioxide-equivalent per megapixel), it is relevant to mention that the 2000 images in the tested dataset contain in total 597.5996 Mpixels. This is a task duty cycle of 0.139% for Scenario 1.

Scenario 2 has approximately the highest possible duty cycle for the ImagePrep task. Instead of running the task once per day, the task is run once every 90 seconds. Since the task takes 87 seconds to complete on average for 18 devices, this is a duty cycle of 96.67%. The equation for CCI for both scenarios is shown below.

$$\text{CCI} = \frac{\mathbb{C}_M + \mathbb{C}_{C+N}}{\sum_{lifetime} \text{ops}} = \frac{\mathbb{C}_M + \text{CI}_{\text{solar}} \times \text{Lifetime} \times (\text{Frac}_{\text{active}} \times \text{Power}_{\text{active}} + \text{Frac}_{\text{sleep}} \times \text{Power}_{\text{sleep}})}{\sum_{lifetime} \text{ops}} \tag{5.2}$$

The parameters used in the CCI calculations are as follows:

- CI_{solar} is the carbon intensity of solar power as stated by Switzer et al., 48 gCO_2-e/kWh. In addition to providing a lower bound on CCI since solar energy has the lowest carbon emissions of the electricity sources identified by Switzer et al., in remote locations solar panels may be the only realistic power source for Cluster.
- Lifetime is $31\text{d} \times 24\frac{\text{h}}{\text{d}} = 744$ hours.
- $\text{Frac}_{\text{active}}$ is the fraction of the total time that the ImagePrep task is running.
- $\text{Frac}_{\text{sleep}}$ is the fraction of time the mobile devices are sleeping (for Scenario 1) or the app is idle (for Scenario 2) when the rest time between runs is too short for the mobile devices to go to sleep and wake up again in time for the next run. For example, Figure 2.7 shows that the connection time for Wi-Fi is over 4 s, so in Scenario 2 the Cluster app must go into idle mode to maintain the Wi-Fi connnection.
- $\text{Power}_{\text{active}}$ is the power consumed while the ImagePrep task is running with 18 devices, 30.71 W, interpolated from the sum of both Android and iOS measurements at 16 and 25 devices, see Figure 5.14.
- $\text{Power}_{\text{sleep}}$ is the power consumption of 18 average mobile devices in sleep mode for Scenario 1, 3.77 W, and the power of 18 devices running the app, but in an idle state for Scenario 2, 22.04 W.
- The sum of lifetime operations for Scenario 1 is $597.6\frac{\text{Mpixels}}{\text{run}} \times 31$ runs $= 18525.5$ Mpixels.

- The sum of lifetime operations for Scenario 2 is $597.6\frac{\text{Mpixels}}{\text{run}} \times 31\text{d} \times 62400\frac{\text{s}}{\text{d}}/90\frac{\text{s}}{\text{run}} = 12844407.4$ Mpixels.

The calculated CCI values for both scenarios are shown below.

$$\text{CCI}_1 = \frac{135.82 \text{ gCO}_2\text{-e}}{18525.5 \text{ Mpixels}} = 7.331 \text{ mgCO}_2\text{-e/Mpixel}$$

$$\text{CCI}_2 = \frac{1086.43 \text{ gCO}_2\text{-e}}{12844407.4 \text{ Mpixels}} = 0.0846 \text{ mgCO}_2\text{-e/Mpixel}$$

Per device, this is $\text{CCI}_1 = 0.407$ mgCO$_2$-e/Mpixel and $\text{CCI}_2 = 0.00470$ mgCO$_2$-e/Mpixel. This is compared to the results from Switzer et al. where CCI was equal to 0.012 mgCO$_2$-e/Mpixel (per Pixel 3A device) and 0.041 mgCO$_2$-e/Mpixel (per Nexus 4 device) for their PDF rendering benchmark. The computations in the ImagePrep task and the PDF rendering benchmark chosen by Switzer et al. are not equivalent, so the CCI calculations for Cluster are simply indications that Cluster has a carbon footprint within an order of magnitude greater (Scenario 1) or smaller (Scenario 2) than the system implemented by Switzer et al. The CCI numbers indicate that Cluster should be more carbon-efficient than utilizing a conventional data center for the same tasks, however this preliminary result must be followed up by more extensive comparative studies to draw significant conclusions.

5.6 Faraday Cage Experiments

Using Wi-Fi to network the Cluster client nodes to the server introduces two issues. First, interference from other devices and wireless LAN transmitters outside the cluster can reduce network performance. Second, hackers could obtain access to the Cluster packets to steal information or sabotage operation, which could have serious consequences if Cluster is employed in a cyber-physical system. While encryption could improve security, the additional overhead could further degrade performance. Isolating the cluster from outside interference and hackers can be accomplished by electromagnetically shielding the wireless network. In addition, one possible use case for Cluster with improved sustainability is to fill a large used metal shipping container with shelves of used mobile devices to create a private database, distributed computer, or other Cluster application. In order to test Cluster's performance in this kind of environment and quantify the effects of shielding the network of devices from the electromagnetic noise of the world, a series of task characterization experiments were run in a large, sealed, walk-in Faraday cage in the basement of a university research building.

The Faraday cage is 1.82 m deep $\times$ 1.90 m wide $\times$ 2.93 m tall and made of 15-centimeter-thick riveted aluminum and steel walls filled with an electromagnetic (EM) radiation absorbing material. The small room is built to building code with a fire sprinkler, active ventilation, electric lights and outlets, and a large metal door similar to a walk-in refrigerator, but the door slips into copper brackets that make the room almost air-tight (if it weren't for the ventilation system) and completely shielded against a large range of EM radiation from radio waves to Wi-Fi and more.

The experiments in the cage were primarily focused on the best location for the Wi-Fi antenna of the router to be located. Three sets of experiment runs were performed on the three tasks that were implemented at the time, ImagePrep, ServerPrep, and DependentCalc. For each experiment, the mobile devices were laid out on 2 inches of corrugated cardboard sitting on top of 4 inches of cushion foam. That setup, the server computer, and the Wi-Fi router were elevated 1.22 meters off the floor using two plastic folding tables. The table with all the mobile devices was placed against one wall so some devices were 3 cm from the wall and some were as far as 60 cm from the wall.

The three locations tested for the Wi-Fi antenna were in the center of the room, against the opposite wall as the devices (4 centimeter clearance to the wall), and in one corner of the room (4 centimeter clearance to both walls). The results of these experiments are shown, along with data of the corresponding tasks from the other timed experiments not in the Faraday cage, in Figure 5.19.

The general results of the experiments in the Faraday cage are that tasks with a high amount of network traffic perform worse in the large metal box than in a more typical wood and drywall room, and tasks that have relatively little network traffic perform equal to the drywall room environment, but with a higher standard deviation in the completion time.

The DependentCalc results are less conclusive, but generally agree with this result. The issues that were most likely responsible for the inconsistent results of the DependentCalc experiments were Cluster app stability, lack of accessibility to the experimental setup, and newness of the DependentCalc task implementation. The DependentCalc task tested in the cage was an early version of the task with less consistent results than the improved version tested later for the timed experiments described in Section 4.2.2. Also, the large metal door to the cage is heavy and requires significant strength and body weight to

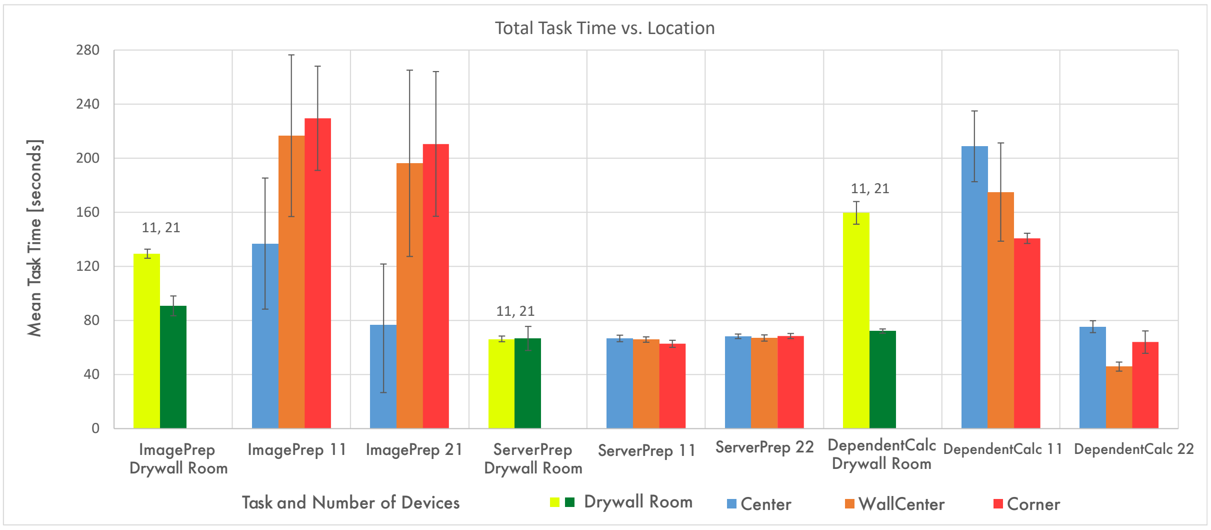


Figure 5.19: Measured Task Completion Time for Faraday Cage Deployment

push the lever down that latches it closed. For this reason, the tasks were left to run unattended for two hours at a time to minimize the operation of the door. This lack of supervision, and the fact that the Cluster app crashed more often in the version that existed when the Faraday cage experiemnts were run, are likely causes of error and variance in the results.

However, the ImagePrep and ServerPrep results are quite clear. The antenna location matters for high network traffic tasks like ImagePrep and doesn't matter for other tasks. The metal walls of the room are highly reflective to EM waves like Wi-Fi and so there is significantly more interference generated by multipath effects during a high wireless traffic task. It can be observed that moving the antenna away from the walls is sufficient to make the Faraday cage a viable location for even a high-traffic task, but the standard deviation is significantly higher in a small metal-enclosed space for such tasks because of corrupted packets and interference. The average completion time is about equal for all tasks in the Faraday cage, particularly if the router antenna is not too close to the metal walls. If there is any benefit to blocking out the EM radiation present in normal office rooms, it seems to be mitigated by the high amount of signal reflections from sheet-metal walls. Further study is needed to see if an anechoic chamber, where there are no reflections off the walls and no outside interference from EM radiation either, might offer some benefit to the higher-traffic tasks or any other task.

5.7 Idle Time and Performance

Although the effects of idle time on the performance of the system were not extensively studied in this research, some observations have been made on the subject as a result of running experiments for 10 – 20 consecutive hours during some weeks. Often the most anomalous results were observed when the mobile devices and the server computer had been running experiments for more than 15 hours without being restarted or allowed to enter sleep mode.

The Cluster app ensures that the mobile devices cannot enter sleep mode while the app is running, regardless of whether a task is running or not. This is to ensure the TCP socket stays connected so that the app can be sent commands at any time from the server. Though the device cannot lock (i.e., enter sleep mode), the Cluster app does allow the screen to display all black, but the screen is not powered-off, even while a task is running. The app was designed to be opened once and then never need user interaction again unless it crashes. The server commands all other actions for the app to take and the app automatically searches for and connects to the server if it is running.

As a result of this design, the mobile device's internal temperature rises after 15 hours of non-sleep operation. The devices were not running tasks continuously during these longer durations of non-sleep, but the devices still shows signs of needing to idle or restart after about 15 hours of being active. The tasks often run slower as a result of the device's processor running at elevated temperature, a result consistent with the literature [87]. Additionally it is well known that software is not perfect and many hours of operation without restart leads to worsening performance because of memory leaks and other small implementation errors. Often, the data collected during this period of continuous Cluster operation was observed to be significantly higher in standard deviation and the tasks would take 200% – 400% longer to complete. Those results were excluded from the data shown in this dissertation. In every instance, the devices were restarted and the results became much more consistent afterwards. Further study of the effect of idle time and thermal management of devices on the performance of a retired-mobile-device-based system is planned for the future.

5.8 Related Work

Chapter 1 presented an overview of how the present work fits in the context of past and current research related to reusing mobile devices. In this section, related work to the research presented in this dissertation is discussed in more detail.

5.8.1 Distributed Sensing

Several research projects have explored ways of reusing mobile devices directly for distributed sensing or as gateways for sensors. For example, the Rainforest Connection organization [17] and Deichmann et al. [18] employed used cellphones to upload audio data from passive bioacoustic monitoring to detect illegal logging and study the impact of natural gas exploration on tropical forest biodiversity, respectively. Maker et al. developed an app that allowed a Nokia N80 smartphone to be used as a camera trap, however they did not deploy and characterize their repurposed devices in the field [19]. Zink et al. characterized a mobile phone repurposed as an in-car parking meter to perform a comparative life cycle assessment of reuse versus refurbishing the phone [11]. The present work focused specifically on characterizing how the different wireless interfaces supported on mobile devices can be used for data transfer, and is therefore complementary to research on repurposing mobile devices as sensors.

Employing retired mobile devices to serve as gateways between IoT devices and the cloud has been proposed by several research groups. The mobile device short-range wireless interfaces (Wi-Fi, Bluetooth Low Energy) would be used to collect data from IoT nodes and the mobile devices would then use their cellular interface (GSM, 5G) to upload data to the cloud. For example, Klugman et al. [14, 15] studied using Android phones as cellular gateways in Zanzibar, Tanzania, to upload AC power measurement data from a commercial off-the-shelf plug-load power meter. The present work focuses instead on how reused mobile devices can use their wireless interfaces to exchange small amounts of data with a query vehicle or data mule, that would then connect to a gateway at a different location. Therefore, the results presented here complement prior work on reusing mobile devices directly as network gateways.

5.8.2 Distributed Computing and Data Storage

Despite initial promise, the concept of building mobile devices out of modular components that could be upgraded and reused has produced disappointing results [12, 13]. Consequently, several publications have explored how entire mobile devices can be reused in public-resource computing, cloud computing, and fog computing. As an example appliction in public-resource computing, Büsching et al. proposed creating ad-hoc distributed computing systems out of collections of physically co-located personal mobile phones, e.g. using all of the personal phones of passengers on a train to compute a local weather forecast [16]. They implemented DroidCluster, a cluster of six LG P500 Android phones with Linux installed alongside Android, and characterized DroidCluster's performance on the LINPACK benchmark running on the MPI library. In contrast, Cluster, the distributed computer presented here, uses a heterogeneous collection of Android and iOS devices, and uses a custom communication protocol instead of MPI between the Cluster nodes. In addition, Cluster was characterized on a variety of tasks related to supporting cyber-physical systems and not just scientific computing. DroidCluster examined performance scaling up to only 6 devices, while the present work studied scaling to 16 to 22 devices, depending on the task.

Shahrad and Wentzlaff [25] propose a server consisting of 84 decommissioned mobile devices, a network router, three rows of fans for cooling, and a power supply, all fitting in the same dimensions as a standard 19-inch 2U rack-mounted server. The proposed server would fit into a data center supporting an Infrastructure-as-a-Service (IaaS) cloud. They proposed interconnecting the decommissioned mobile devices (Samsung Galaxy Note 4) with USB cables and hubs. They also completed a total cost of ownership (TCO) analysis that compares their proposed server with a standard rack-mounted server with similar performance. However, they did not construct or experimentally characterized their proposed server. The present work instead built and experimentally characterized a cluster of wirelessly-connected heterogeneous devices with an emphasis on supporting small workloads characteristic of cyber-physical systems rather than generic workloads supported in IaaS clouds. Switzer et al., in a project they call Junkyard Computing, demonstrated that a collection of ten three-year-old smartphones could provide good performance on IaaS cloud microservice benchmarks and quantified how reusing mobile devices as cloud infrastructure could reduce the carbon footprint of the mobile devices [26]. Their testbed consisted of 10 Google Pixel 3A and Pixel 3A XL phones connected using Wi-Fi similar to the network connectivity of Cluster. However, Switzer et al. replaced the native Android OS with Ubuntu Touch, an open-source mobile OS that supports a desktop-like user experience, and added kernel modifications to support Docker, which was needed to run instances of the cloud benchmarks they used to compare their performance to Amazon Web Services EC2 C5 instances. A significant limitation of this approach is that Ubuntu Touch supports only 63 different device models as of September 2023 (https://devices.ubuntu-touch.io/), including no support for Apple devices. The Cluster software presented here, built as an app on top of the mobile device's native OS, can be run on a vast number of different mobile device models with no user modifications other than downloading and installing the app. The Cluster tasks differ from the Junkyard Computing applications in that the Cluster tasks are smaller-scale and target the cyber-physical systems domain, rather than consumer cloud microservices. The Cluster hardware scaled to 16 to 22 devices, exceeding the 10 devices used in the Switzer et al. implementation [26].

From the viewpoint of a distributed computing architecture based on reused mobile devices, the Renée project is the most closely-related prior work to the present research [27]. Renée demonstrated that a bank of four retired phones (Google Nexus 4) connected by Wi-Fi and managed by a single central computer could function as a small-scale data center (cloudlet) or fog node to provide Function-as-a-Service (FaaS)/Platform-as-a-Service (PaaS) capabilities. Similar to Switzer et al. [26], the native Android OS was replaced with Ubuntu Touch with a modified kernel. Users submit jobs to Renée's central manager computer (a Raspberry Pi), which then distributes the job to one of the mobile devices. The format of the jobs is a zipped Linux folder containing an executable shell script and associated programs and data, similar to the format required by Amazon Lambda instances. The performance of Renée was characterized on four microbenchmarks that include numerical computation, data analytics, image processing, and machine learning on a small data set. A major difference between Renée and Cluster is Renée's reliance on a homogeneous set of mobile devices. Unlike their system, Cluster was designed to use any iOS device that runs iOS 10.0.0 or higher (iPhone 5 or newer, iPad 4th gen or newer) or any Android phone or tablet that runs Android 5.0 Lollipop (API 21) or newer (e.g. Nexus 5 or newer). The Cluster hardware scaled to 16 to 22 mobile devices, exceeding the 4 devices used in the Renée implementation. In contrast to both Junkyard Computing [26] and Renée [27], Cluster's wireless network performance was studied in both open and shielded electromagnetic environments.

The FAWN project [28, 29] inspired much later work exploring how coupling low-power processors with solid-state data storage could be used to create energy-efficient data center computing. FAWN constructed a system using 21 commodity single-board computers with flash-based storage and designed a custom data store and key-value lookup system optimized for flash. In the present work, the FAWN data store and lookup architecture was ported to Cluster, whose "wimpy" nodes consist of heterogeneous retired mobile devices and their built-in solid-state storage, and its performance was characterized. Cluster is the only implementation of a distributed data store on reused mobile devices that the author is aware of.

Chapter 6

Conclusions and Future Work

This chapter discusses conclusions that can be drawn at this time from the presented research and plans to expand and continue this research in the future. The field of distributed sensing and distributed computing with retired mobile devices is fairly new and is already providing exciting results.

6.1 Distributed Sensing

As discussed in Chapter 1, retired mobile devices have been reused as distributed sensors for bioacoustic monitoring and as network gateways for IoT applications; in both of those previous works, the devices' cellular connections were used to communicate sensor data. The present work explores how five different wireless interfaces besides the cellular interface can be used to communicate data from a mobile device reused as a sensor to a query vehicleor data mule. For query vehicle networks with a similar topology to that studied in this work, Wi-Fi is the best choice of radio scheme. The transmission rate is faster than the other radio schemes and the total transaction time is shorter for more than 30 kB of data. With Wi-Fi radios being available for $4 or less and many microcontroller modules like the ESP32 used in this work having Wi-Fi built-in, the cost of adding Wi-Fi connectivity if necessary is very low. Effectively all retired mobile devices that can be reused for distributed sensing already incorporate Wi-Fi, so application in a query vehiclewireless sensor network is another use case that motivates the reuse of mobile devices. Last, Wi-Fi is specifically designed to be able to stream data from multiple connected devices, much more so than other radio schemes studied here, and enabling multiple simultaneous connections between the query vehicleand the distributed sensors will make query vehiclepath planning and scheduling significantly more efficient.

6.2 Distributed Computing and Data Storage

To explore the reuse of retired mobile devices in cyber-physical systems for distributed computing and data storage, the present work developed Cluster: a distributed computing system composed of a heterogeneous collection of retired smartphones and tablets. These are mobile devices that are used, at the typical end of their consumer lifespan (retired), but are still functional, and would otherwise be recycled for their raw materials. The abstracted design of the Cluster software is intended to simplify the process of adding new tasks to the library of applications Cluster is capable of running. As the implementation of four tasks for this research were finished, this design proved to be successful at those intentions. Each layer of the software's structure is separate enough to allow new tasks to be added without becoming entangled in the existing code or affecting the performance of any existing task. This design makes it possible for virtually any task and any number of tasks to be added to the Cluster software in the future in a modular way.

With the design of the Cluster server and Cluster app software successful in its goals of modularity and abstraction, the Cluster characterization experiments were able to proceed quickly enough to draw some early conclusions about the nature of a system like Cluster. First, the limitations of an architecture like Cluster became clear. Devices in the cluster rely on Wi-Fi as the sole means of communicating with the rest of the network. This is different from how data centers, supercomputers, parallel multicore processors, and most desktop-based distributed computers operate in that they have wired connections between compute nodes. Wi-Fi uses more power and has more limited data rates than a wired connection. Wi-Fi also becomes less reliable as more devices are communicating in a space at the same time because of collisions between multiple Wi-Fi transmitters sharing the same wireless channel. Interference from Wi-Fi users that are not part of the cluster is also possible.

While enclosing Cluster in a electromagnetically-shielded space (i.e. within a metal Faraday cage), such as a used shipping container, does not show any improvement in the performance of any specific task, it may still be possible to negate some of the effects of increasing amounts of wireless interference as the number of devices grows by using a less reflective Faraday cage such as an anechoic chamber. At the very least, the performance of low-network-traffic tasks is not hindered by the environment of a solid metal Faraday cage and would provide isolation from possible outside interference or hacking by malicious actors without suffering any performance losses.

Additional limitations evident from conducting the Cluster characterization experiments are the lack of compatibility with any existing programs for doing parallel or distributed computing. Existing applications must be rewritten (ported) to be compatible with Cluster. This does not make the architecture any less viable as a computing resource, but does limit how fast any application can be deployed on the cluster. Previous work has uniformly focused on replacing the native mobile OS on reused devices with a Linux operating system and then installing Linux applications. While this approach reduces the barrier to running different applications on a cluster of reused mobile devices, it severely limits the number of devices that could be successfully reused since very few models are supported by available mobile Linux operating systems.

Finally, the scalability of the Cluster design is limited to $\sim 20 - 200$ devices depending on the network utilization and the server CPU utilization demanded of the task being run. This scalability limit can be overcome with a redesign of the system to allow multiple servers, multiple Wi-Fi routers, and the ability for client nodes to send data directly to other client nodes instead of going through a central server node.

Despite the limitations, Cluster's performance on tasks such as batch pre-processing of images and implementing a key-value data store is sufficient to support many functions of a cyber-physical system, and there are some applications where Cluster excels over alternatives. This can be determined from the power experiments alone. The fact that Cluster uses less power than almost any other distributed computer, because the manufacturer prioritized low-power and small size in the design of these mobile devices, implies that there are some applications in which Cluster will accomplish the same computation with less energy cost than any other system. Similarly, the small size of the Cluster network, which requires only one Wi-Fi router per ~ 50 devices, and one server node per $\sim 20 - 200$ client nodes depending on the task, and no other networking or cooling hardware implies that Cluster is also more space-efficient than almost any other distributed computer. The limited number of Cluster nodes is still sufficient for supporting fog computing and moving some data center functions closer to the edge nodes of a cyber-physical system to reduce latency. Furthermore, the compute nodes cost from \$0 to \$30 on average because they are retired devices with low perceived value, and can be obtained from the used electronics market or donated to a particular Cluster implementation.

In addition to these advantages, the environmental benefits of reusing the hardware of the mobile devices, up to 20 times more beneficial than recycling the materials, and the saved environmental and financial cost of manufacturing new devices, makes Cluster a more environmentally-friendly alternative to almost any other computing system. These factors are enough to conclude that an important component of the future of affordable and efficient distributed computing, data storage, and embedded-systems-based sensing is undoubtedly retired mobile devices. Smartphone manufacturers designed the devices to be powerful low-power computers. The benefits of using these devices after their intended consumer lifetime would be even greater if they were also designed to accept a Linux operating system that has full access to all computing resources and is capable of establishing a socket connection through the charging cable instead of the Wi-Fi interface. Then the major limitations, stability concerns of apps crashing and being limited in their capabilities when the app is not in the foreground of the operating system in addition to the slow, non-scalable, and high-power nature of Wi-Fi, would be eliminated.

6.3 Future Work

Section 2.9 describes future work related to extending the wireless network transaction characterization study. This section will focus on future work related to Cluster.

After extensive efforts on researching and implementing the Cluster distributed computer architecture, there are still many more research questions to be answered and experiments to follow up on. The research questions to investigate in the future, with suggested or planned experiments to answer them, are as follows:

1. Is the performance and energy consumption of the mobile device cluster comparable to a conventional implementation of inexpensive and simple cluster computing, a network of Raspberry Pi computers? How does the Computational Carbon Intensity (CCI) compare for identical computational tasks?

   Implement the same computing tasks on a cluster of Raspberry Pi computers and benchmark it using the same standards on which Cluster has been evaluated so far.

2. Does using multiple processes instead of a single multithreaded process on the server program improve the scalability and how does it affect performance?

   This has been partially studied, but not thoroughly investigated. The multithreaded C# version of the server program must be updated with the latest task implementation improvements from the Python version. Each task must be characterized with both versions using multiple processes vs. a single process to investigate the benefits of using multiple processes. Additionally it is planned to compare the performance of running the Python server program as an executable instead of interpreted in real time.

3. What is the limit of how many devices can operate in Cluster until performance no longer improves?

   This does depend on what kind of task is being performed, but in general the scalability of the system is limited to about 25 – 30 devices with the current design. Wi-Fi routers are not designed to handle more than 100 – 200 devices (the number of IP addresses that they will assign is limited to 255 per wireless LAN). Larger LANs do exist for offices and universities, so using a system like those used in larger networks would address scalability in terms of the network hardware. However, being able to coordinate that many devices would require a re-design as a single server computer could not handle communications with more than about 30 devices as the present Cluster research shows. A system where client nodes can send data directly to each other (similar to how MPI works, but still using an app-based approach) and multiple server nodes share the coordination effort would be necessary to scale beyond 200 reused mobile devices.

4. What is the reliability (mean time to failure, e.g. due to Cluster app crashes) of a system built on running an app on devices that are not designed to run a single app for more than a couple of hours?

   One suggested approach to estimating the mean time to failure is to run each of the tasks for a whole day or longer, with the server logging which nodes disconnect and when, until time is up or all apps have crashed.

5. Can the mean time to failure be compensated for by programming the Cluster app to automatically be restarted when it crashes?

   There is a special developer tool from Apple called Apple Configurator that can install apps to multiple managed devices simultaneously and put devices into "kiosk mode", a state where an iOS device is locked into running one or two apps from the instant kiosk mode is entered until the device is taken out of kiosk mode. This mode could be used to automatically restart the app that crashes and keep manual maintenance of each device to a minimum. Android devices have a similar kiosk mode meant for display or public demonstration purposes.

6. How is the performance affected by letting each device idle periodically while performing computationally-intense or network-intense tasks?

It is suggested to change the MainLoopDelay built into each task and measure the performance again. This may also affect the mean time to failure, so that experiment should also be re-run. Another suggestion is to build in a timer in the Cluster app that periodically disconnects the app from the server and then reconnects after a minute of idle time. This simple disconnect should also clear up quite a few memory leaks as many threads in the app will be closed as a result of disconnecting from the server.

7. How is the performance of Cluster affected by further changing its environment?

   Repeating the Faraday cage experiment in a typical wood frame and drywall room, but varying the size of the room, the surface the devices are sitting on, the distance devices are placed from each other, and the distance from the walls. Also, it is suggested to repeat the characterization experiments in an anechoic chamber, which is hypothesized to reduce the multipath interference compared to reflective walls in the room where the cluster is placed.

8. How does Cluster perform as a RAID storage implementation compared to other published works on RAID?

   The performance of Cluster on the FAWN key-value data store suggests that Cluster could also perform well as a solid-state disk array. A suggested study is to implement at least the three most common levels of the RAID protocol with each mobile device emulating one "hard drive". The study should test read and write speed and test error correcting in the same manner as RAID implementations are typically evaluated.

9. What is the maximum read and write speed for a FAWN or RAID implementation?

   For testing read speed, this experiment would utilize the ability to fetch all data in the database of a mobile app as fast as possible and then report those statistics back to the server instead of sending each entry as they are fetched. For write speed, the app itself would generate a large amount of data first and then store it all at once instead of waiting for each entry to arrive from the server node before storing it.

10. For dependent calculations, how is performance affected by increasing the number and widening the age distribution of neighboring nodes which a client node must get data from in order to continue?

It is likely that a new bottleneck exists when a higher percentage of the network traffic and client idle time is from an increase in the dependent data being passed between nodes. This may happen in two ways: either the computation requires passing data more often, i.e. the period between checkpoints is shorter, or by increasing the number of neighbors required for each checkpoint exchange. In addition, older, slower devices are likely to emerge as bottlenecks as newer, faster devices wait on their results. An additional improvement to the synthetic dependent calculation task is to make the client nodes send their data directly to their neighbors instead of going through the server node. Ultimately, the synthetic task should be complemented by a task relevant to a real cyber-physical system.

11. How can a model with predictive power of systems utilizing retired mobile devices like Cluster be developed to aid system designers in constructing and provisioning these systems for use in various applications?

    Queuing theory shows promise as an approach to developing a quantitative performance model for Cluster and similar systems, as it has been used in previous work to model performance for cloud infrastructure [108]. It is suggested to first develop a queuing theoretic model for Cluster and calibrate it with the results of the Cluster characterization experiments presented in this work. The next step would be to validate the model against experimental data derived from characterizing alternative Cluster implementations with different numbers and types of devices.

The capabilities and benefits of distributed computing systems made from retired mobile devices are evident from this research, and yet it is also clear that there is much more to be learned, characterized, and improved about such systems.

Appendix A

Links and Acknowledgments

A.1 Experimental Data

The raw data collected for the experiments in this research can be found at the following links:
Transaction Characterization data: https://ucdavis.box.com/s/cpb23khvz3v7znnllmjxcqp2flq7ucg1
Cluster data: https://ucdavis.box.com/s/f53fg0vr5dne2lsuhbmk8pv5m6ov2jjy

A.2 Acknowledgments

Below is a list of acknowledgments and special recognition.

- Professor Raj Amirtharajah for years of help with difficult engineering problems and writing, and for close advising every step of the way.
- The Qualifying Exam committee for much helpful advice about focusing this research into a manageable and ultimately successful plan: Venkatesh Akella, Raj Amirtharajah, Stavros G. Vougioukas, Hooman Rashtian, Isaya Kisekka
- Professor William Putnam for making the Faraday cage available for six full days of experiments.
- The Dissertation committee for advice about the final writing for this document.

A.2.1 Donors

Tim Ambrose used some of his own retired devices for this work. Special thanks to those who donated their used devices to the Cluster project:
Matt Ambrose, Brooke Ambrose, Denise Monahan, Annika Peterson, Afton Geil, Raj Amirtharajah, John Squires, and Bob Hodash

Bibliography

[1] The Radicati Group, “Forecast number of mobile devices worldwide from 2020 to 2025 (in billions) [Graph].” https://www.statista.com/statistics/245501/multiple-mobile-device-ownership-worldwide/, April 7, 2021. [Online]. Accessed August 28, 2023.

[2] Ericsson, “Number of smartphone mobile network subscriptions worldwide from 2016 to 2022, with forecasts from 2023 to 2028 (in millions) [Graph].” https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/, June 1, 2023. [Online]. Accessed August 28, 2023.

[3] World Economic Forum, “Projected electronic waste generation worldwide from 2019 to 2030 (in million metric tons) [Graph].” https://www.statista.com/statistics/1067081/generation-electronic-waste-globally-forecast/, July 1, 2020. [Online]. Accessed August 30, 2023.

[4] F. Richter, “Smartphones: Aging like wine or milk? [Digital Image].” https://www.statista.com/chart/26687/smartphone-price-depreciation/, January 26, 2022. [Online]. Accessed September 5, 2023.

[5] StatCounter, “Mobile operating systemsḿarket share worldwide from 1st quarter 2009 to 2nd quarter 2023 [Graph].” https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/, July 1, 2023. [Online]. Accessed August 28, 2023.

[6] L. Acaroglu, “Where do old cellphones go to die?,” *The New York Times*, May 5, 2013.

[7] M. Cordella, F. Alfieri, and J. Sanfelix, “Reducing the carbon footprint of ICT products through material efficiency strategies: A life cycle analysis of smartphones,” *Journal of Industrial Ecology*, vol. 25, no. 2, pp. 448–464, 2021.

[8] M. Ercan, J. Malmodin, P. Bergmark, E. Kimfalk, and E. Nilsson, “Life cycle assessment of a smartphone,” in *Proceedings of ICT for Sustainability 2016*, pp. 124–133, Atlantis Press, 2016/08.

[9] J. Y. Oliver, R. Amirtharajah, V. Akella, R. Geyer, and F. T. Chong, “Life cycle aware computing: Reusing silicon technology,” *Computer*, vol. 40, no. 12, pp. 56–61, 2007.

[10] I. Ilankoon, Y. Ghorbani, M. N. Chong, G. Herath, T. Moyo, and J. Petersen, “E-waste in the international context – a review of trade flows, regulations, hazards, waste management strategies and technologies for value recovery,” *Waste Management*, vol. 82, pp. 258–275, 2018.

[11] T. Zink, F. Maker, R. Geyer, R. Amirtharajah, and V. Akella, “Comparative life cycle assessment of smartphone reuse: Repurposing vs. refurbishment,” *The International Journal of Life Cycle Assessment*, vol. 19, no. 5, pp. 1099–1109, 2014.

[12] M. Brannon, P. Graeter, D. Schwartz, and J. R. Santos, “Reducing electronic waste through the development of an adaptable mobile device,” in *2014 Systems and Information Engineering Design Symposium (SIEDS)*, pp. 57–62, 2014.

[13] S. Byford, “Google reportedly cancels Project Ara modular smartphone plans,” *The Verge*, September 1, 2016.

[14] N. Klugman, M. Clark, P. Pannuto, and P. Dutta, “Android resists liberation from its primary use case,” in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, MobiCom ’18, (New York, NY, USA), p. 849–851, Association for Computing Machinery, 2018.

[15] N. Klugman, V. Jacome, M. Clark, M. Podolsky, P. Pannuto, N. Jackson, A. S. Nassor, C. Wolfram, D. Callaway, J. Taneja, and P. Dutta, “Experience: Android resists liberation from its primary use case,” in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, MobiCom ’18, (New York, NY, USA), p. 545–556, Association for Computing Machinery, 2018.

[16] F. Büsching, S. Schildt, and L. Wolf, "Droidcluster: Towards smartphone cluster computing – the streets are paved with potential computer clusters," in *2012 32nd International Conference on Distributed Computing Systems Workshops*, pp. 114–117, 2012.

[17] M. Ives, "Using old cellphones to listen for illegal loggers," *The New York Times*, 2019.

[18] J. L. Deichmann, A. Hernández-Serna, J. A. Delgado C., M. Campos-Cerqueira, and T. M. Aide, "Soundscape analysis and acoustic monitoring document impacts of natural gas exploration on biodiversity in a tropical forest," *Ecological Indicators*, vol. 74, pp. 39–48, 2017.

[19] F. Maker, R. Amirtharajah, and V. Akella, "Runtime adaptation of applications using design of experiments: A smartphone-based case study," *IEEE Embedded Systems Letters*, vol. 6, no. 2, pp. 25–28, 2014.

[20] The RainforestConnection, "The Rainforest Connection Guardian Platform." https://rfcx.org/guardian, 2023. Accessed August 28, 2023.

[21] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum, "Programming languages for distributed computing systems," *ACM Comput. Surv.*, vol. 21, p. 261–322, sep 1989.

[22] D. Anderson, "BOINC: a system for public-resource computing and storage," in *Fifth IEEE/ACM International Workshop on Grid Computing*, pp. 4–10, 2004.

[23] P. Mell and T. Grance, "The NIST definition of cloud computing," NIST Special Publication (NIST SP) 800-145, National Institute of Standards and Technology (NIST), U.S. Dept. of Commerce, Gaithersburg, MD, USA, Sept. 2011. Accessed August 28, 2023.

[24] M. Iorga, L. Feldman, R. Barton, M. Martin, N. Goren, and C. Mahmoudi, "Fog computing conceptual model," NIST Special Publication (NIST SP) 500-325, National Institute of Standards and Technology (NIST), U.S. Dept. of Commerce, Gaithersburg, MD, USA, 2018. Accessed August 24, 2023.

[25] M. Shahrad and D. Wentzlaff, "Towards deploying decommissioned mobile devices as cheap energy-efficient compute nodes," in *9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 17)*, 2017.

[26] J. Switzer, G. Marcano, R. Kastner, and P. Pannuto, "Junkyard computing: Repurposing discarded smartphones to minimize carbon," in *Proceedings of the 28th ACM International Conference on*

Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, (New York, NY, USA), p. 400–412, Association for Computing Machinery, 2023.

[27] J. Switzer, E. Siu, S. Ramesh, R. Hu, E. Zadorian, and R. Kastner, "Renée: New life for old phones," *IEEE Embedded Systems Letters*, vol. 14, no. 3, pp. 135–138, 2022.

[28] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "Fawn: A fast array of wimpy nodes," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, (New York, NY, USA), p. 1–14, Association for Computing Machinery, 2009.

[29] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "Fawn: A fast array of wimpy nodes," *Commun. ACM*, vol. 54, p. 101–109, jul 2011.

[30] T. Ambrose, V. Akella, and R. Amirtharajah, "Wireless transaction time characterization for query vehicle networks," in preparation.

[31] R. Sanchez-Iborra, J. Sanchez-Gomez, J. Ballesta-Viñas, M.-D. Cano, and A. F. Skarmeta, "Performance evaluation of lora considering scenario conditions," *Sensors*, vol. 18, no. 3, p. 772, 2018.

[32] B. S. I. Group, *Specification of Bluetooth System*, November 2003.

[33] O. Tekdas, V. Isler, J. H. Lim, and A. Terzis, "Using mobile robots to harvest data from sensor fields," *IEEE Wireless Communications*, vol. 16, no. 1, pp. 22–28, 2009.

[34] D. Palma, A. Zolich, Y. Jiang, and T. A. Johansen, "Unmanned aerial vehicles as data mules: An experimental assessment," *IEEE Access*, vol. 5, pp. 24716–24726, 2017.

[35] M. Raj, N. Li, D. Liu, M. Wright, and S. K. Das, "Using data mules to preserve source location privacy in wireless sensor networks," *Pervasive and Mobile Computing*, vol. 11, pp. 244 – 260, 2014.

[36] M. M. Coutinho, A. Efrat, T. Johnson, A. Richa, and M. Liu, "Healthcare supported by data mule networks in remote communities of the amazon region," *International scholarly research notices*, vol. 2014, 2014.

[37] J. Crowcroft, L. Levin, and M. Segal, "Using data mules for sensor network data recovery," *Ad Hoc Networks*, vol. 40, pp. 26 – 36, 2016.

[38] P. B. Val, M. G. Valls, and M. B. Cuñado, "A simple data-muling protocol," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 2, pp. 895–902, 2013.

[39] R. Sugihara and R. K. Gupta, "Data mule scheduling in sensor networks: Scheduling under location and time constraints," *UCSD Tech. Rep.*, 2007.

[40] R. Sugihara and R. K. Gupta, "Scheduling under location and time constraints for data collection in sensor networks," in *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, Citeseer, 2007.

[41] G. Citovsky, J. Gao, J. S. Mitchell, and J. Zeng, "Exact and approximation algorithms for data mule scheduling in a sensor network," in *International Symposium on Algorithms and Experiments for Wireless Sensor Networks*, pp. 57–70, Springer, 2015.

[42] R. Mukherjee, S. Roy, and A. Das, "Survey on data collection protocols in wireless sensor networks using mobile data collectors," in *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, pp. 632–636, IEEE, 2015.

[43] D. Jea, A. Somasundara, and M. Srivastava, "Multiple controlled mobile elements (data mules) for data collection in sensor networks," in *Distributed Computing in Sensor Systems* (V. K. Prasanna, S. S. Iyengar, P. G. Spirakis, and M. Welsh, eds.), (Berlin, Heidelberg), pp. 244–257, Springer Berlin Heidelberg, 2005.

[44] R. Xu, H. Dai, Z. Jia, M. Qiu, and B. Wang, "A piecewise geometry method for optimizing the motion planning of data mule in tele-health wireless sensor networks," *Wireless networks*, vol. 20, no. 7, pp. 1839–1858, 2014.

[45] R. Sugihara and R. K. Gupta, "Optimal speed control of mobile node for data collection in sensor networks," *IEEE Transactions on Mobile Computing*, vol. 9, no. 1, pp. 127–139, 2009.

[46] J.-S. Liu, S.-Y. Wu, and K.-M. Chiu, "Path planning of a data mule in wireless sensor network using an improved implementation of clustering-based genetic algorithm," in *2013 IEEE Symposium on Computational Intelligence in Control and Automation (CICA)*, pp. 30–37, IEEE, 2013.

[47] R. Sugihara and R. K. Gupta, “Improving the data delivery latency in sensor networks with controlled mobility,” in *International Conference on Distributed Computing in Sensor Systems*, pp. 386–399, Springer, 2008.

[48] A. Wichmann, J. Chester, and T. Korkmaz, “Smooth path construction for data mule tours in wireless sensor networks,” in *2012 IEEE Global Communications Conference (GLOBECOM)*, pp. 86–92, IEEE, 2012.

[49] D. Kim, R. Uma, B. H. Abay, W. Wu, W. Wang, and A. O. Tokuta, “Minimum latency multiple data mule trajectory planning in wireless sensor networks,” *IEEE Transactions on Mobile Computing*, vol. 13, no. 4, pp. 838–851, 2013.

[50] R. Sugihara and R. K. Gupta, “Speed control and scheduling of data mules in sensor networks,” *ACM Transactions on Sensor Networks (TOSN)*, vol. 7, no. 1, pp. 1–29, 2010.

[51] R. Sugihara and R. K. Gupta, “Path planning of data mules in sensor networks,” *ACM Transactions on Sensor Networks (TOSN)*, vol. 8, no. 1, pp. 1–27, 2011.

[52] K. Li, C.-C. Shen, and G. Chen, “Energy-constrained bi-objective data muling in underwater wireless sensor networks,” in *The 7th IEEE International Conference on Mobile Ad-hoc and Sensor Systems (IEEE MASS 2010)*, pp. 332–341, IEEE, 2010.

[53] S.-Y. Wu and J.-S. Liu, “Evolutionary path planning of a data mule in wireless sensor network by using shortcuts,” in *2014 IEEE Congress on Evolutionary Computation (CEC)*, pp. 2708–2715, IEEE, 2014.

[54] K. L.-M. Ang, J. K. P. Seng, and A. M. Zungeru, “Optimizing energy consumption for big data collection in large-scale wireless sensor networks with mobile collectors,” *IEEE Systems Journal*, vol. 12, no. 1, pp. 616–626, 2017.

[55] Y.-L. Lai, J.-R. Jiang, *et al.*, “A genetic algorithm for data mule path planning in wireless sensor networks,” *Appl. Math*, vol. 7, no. 1, pp. 413–419, 2013.

[56] J. Ansari, D. Pankin, and P. Mähönen, “Radio-triggered wake-ups with addressing capabilities for extremely low power sensor network applications,” *International Journal of Wireless Information Networks*, vol. 16, no. 3, p. 118, 2009.

[57] S. J. Marinkovic and E. M. Popovici, "Nano-power wireless wake-up receiver with serial peripheral interface," *IEEE Journal on selected areas in communications*, vol. 29, no. 8, pp. 1641–1647, 2011.

[58] G. U. Gamm, M. Sippel, M. Kostic, and L. M. Reindl, "Low power wake-up receiver for wireless sensor nodes," in *2010 Sixth International Conference on Intelligent Sensors, Sensor Networks and Information Processing*, pp. 121–126, IEEE, 2010.

[59] V. Jelicic, M. Magno, D. Brunelli, V. Bilas, and L. Benini, "Analytic comparison of wake-up receivers for wsns and benefits over the wake-on radio scheme," in *Proceedings of the 7th ACM workshop on Performance monitoring and measurement of heterogeneous wireless and wired networks*, pp. 99–106, 2012.

[60] R. Piyare, A. L. Murphy, C. Kiraly, P. Tosato, and D. Brunelli, "Ultra low power wake-up radios: A hardware and networking survey," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2117–2157, 2017.

[61] R. Corvaja, "Qos analysis in overlay bluetooth-wifi networks with profile-based vertical handover," *IEEE Transactions on Mobile Computing*, vol. 5, no. 12, pp. 1679–1690, 2006.

[62] N. Chhabra, "Comparative analysis of different wireless technologies," *International Journal Of Scientific Research In Network Security & Communication*, vol. 1, no. 5, pp. 3–4, 2013.

[63] J.-S. Lee, Y.-W. Su, and C.-C. Shen, "A comparative study of wireless protocols: Bluetooth, uwb, zigbee, and wi-fi," in *IECON 2007-33rd Annual Conference of the IEEE Industrial Electronics Society*, pp. 46–51, Ieee, 2007.

[64] M. Duarte, C. Dick, and A. Sabharwal, "Experiment-driven characterization of full-duplex wireless systems," *IEEE Transactions on Wireless Communications*, vol. 11, no. 12, pp. 4296–4307, 2012.

[65] S. Jain, R. C. Shah, W. Brunette, G. Borriello, and S. Roy, "Exploiting mobility for energy efficient data collection in wireless sensor networks," *Mobile networks and Applications*, vol. 11, no. 3, pp. 327–339, 2006.

[66] G. Anastasi, M. Conti, E. Monaldi, and A. Passarella, "An adaptive data-transfer protocol for sensor networks with data mules," in *2007 IEEE international symposium on a world of wireless, mobile and multimedia networks*, pp. 1–8, IEEE, 2007.

[67] G. Anastasi, M. Conti, E. Gregori, C. Spagoni, and G. Valente, “Motes sensor networks in dynamic scenarios: an experimental study for pervasive applications in urban environments,” *Journal of Ubiquitous Computing and Intelligence*, vol. 1, no. 1, pp. 9–16, 2007.

[68] G. Anastasi, M. Conti, and M. Di Francesco, “Data collection in sensor networks with data mules: An integrated simulation analysis,” in *2008 IEEE Symposium on Computers and Communications*, pp. 1096–1102, IEEE, 2008.

[69] D. Ridge, D. Becker, P. Merkey, and T. Sterling, “Beowulf: harnessing the power of parallelism in a pile-of-PCs,” in *1997 IEEE Aerospace Conference*, vol. 2, pp. 79–91 vol.2, 1997.

[70] J. Kiepert, “Creating a Raspberry Pi-based Beowulf cluster,” 2013.

[71] T. More, “Thinking about selling your phone?.” https://www.trademore.com/sell/how-much-is-my-phone-worth, 2021.

[72] I. LLC, “Trade-in your uses electronics.” https://www.itsworthmore.com/, 2023.

[73] M. Deruyck, W. Vereecken, E. Tanghe, W. Joseph, M. Pickavet, L. Martens, and P. Demeester, “Comparison of power consumption of mobile wimax, hspa and lte access networks,” in *2010 9th Conference of Telecommunication, Media and Internet*, pp. 1–7, IEEE, 2010.

[74] J. Baliga, R. Ayre, K. Hinton, and R. S. Tucker, “Energy consumption in wired and wireless access networks,” *IEEE Communications Magazine*, vol. 49, no. 6, pp. 70–77, 2011.

[75] F. Nielsen, *Introduction to MPI: The Message Passing Interface*, pp. 21–62. 02 2016.

[76] P. Bright, “Google going its own way, forking webkit rendering engine,” *Ars Technica*, April 2013. Retrieved June 2020.

[77] T. Li, V. K. Narayana, E. El-Araby, and T. El-Ghazawi, “Gpu resource sharing and virtualization on high performance computing systems,” in *2011 International Conference on Parallel Processing*, pp. 733–742, 2011.

[78] E. A. Lee, “The problem with threads,” *Computer*, vol. 39, no. 5, pp. 33–42, 2006.

[79] S. Kleiman, D. Shah, and B. Smaalders, *Programming with threads.* Sun Soft Press Mountain View, 1996.

[80] A. Rehman and T. Saba, "Neural networks for document image preprocessing: state of the art," *Artificial Intelligence Review*, vol. 42, pp. 253–273, 2014.

[81] K. De Raad, K. A. van Garderen, M. Smits, S. R. van der Voort, F. Incekara, E. Oei, J. Hirvasniemi, S. Klein, and M. P. Starmans, "The effect of preprocessing on convolutional neural networks for medical image segmentation," in *2021 IEEE 18th International Symposium on Biomedical Imaging (ISBI)*, pp. 655–658, IEEE, 2021.

[82] P. Hurtik, V. Molek, and J. Hula, "Data preprocessing technique for neural networks based on image represented by a fuzzy function," *IEEE Transactions on Fuzzy Systems*, vol. 28, no. 7, pp. 1195–1204, 2019.

[83] D. Makovoz, T. Roby, I. Khan, and H. Booth, "Mopex: a software package for astronomical image processing and visualization," in *Advanced Software and Control for Astronomy*, vol. 6274, pp. 93–102, SPIE, 2006.

[84] A. M. Koekemoer, H. Aussel, D. Calzetti, P. Capak, M. Giavalisco, J.-P. Kneib, A. Leauthaud, O. Le Fevre, H. McCracken, R. Massey, *et al.*, "The cosmos survey: Hubble space telescope advanced camera for surveys observations and data processing," *The Astrophysical Journal Supplement Series*, vol. 172, no. 1, p. 196, 2007.

[85] V. Fadeyev, G. Aldering, and S. Perlmutter, "Improvements to the image processing of hubble space telescope nicmos observations with multiple readouts1," *Publications of the Astronomical Society of the Pacific*, vol. 118, no. 844, p. 907, 2006.

[86] I. Bird, "Computing for the large hadron collider," *Annual Review of Nuclear and Particle Science*, vol. 61, pp. 99–118, 2011.

[87] O. Sahin and A. K. Coskun, "On the impacts of greedy thermal management in mobile devices," *IEEE Embedded Systems Letters*, vol. 7, no. 2, pp. 55–58, 2015.

[88] Apple, "Core data: Persist or cache data on a single device, or sync data to multiple devices with cloudkit." https://developer.apple.com/documentation/coredata, 2023.

[89] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, “Dague: A generic distributed dag engine for high performance computing,” *Parallel Computing*, vol. 38, no. 1-2, pp. 37–51, 2012.

[90] L. Chen, Y. Ma, P. Liu, J. Wei, W. Jie, and J. He, “A review of parallel computing for large-scale remote sensing image mosaicking,” *Cluster Computing*, vol. 18, pp. 517–529, 2015.

[91] I. Raicu, I. T. Foster, and Y. Zhao, “Many-task computing for grids and supercomputers,” in *2008 workshop on many-task computing on grids and supercomputers*, pp. 1–11, IEEE, 2008.

[92] S. Gelly, J.-B. Hoock, A. Rimmel, O. Teytaud, *et al.*, “On the parallelization of monte-carlo planning,” in *ICINCO*, 2008.

[93] L. Maigne, D. Hill, P. Calvat, V. Breton, R. Reuillon, D. Lazaro, Y. Legre, and D. Donnarieix, “Parallelization of monte carlo simulations and submission to a grid environment,” *Parallel processing letters*, vol. 14, no. 02, pp. 177–196, 2004.

[94] D. Takahashi, “Parallel implementation of multiple-precision arithmetic and 2,576,980,370,000 decimal digits of π calculation,” *Parallel computing*, vol. 36, no. 8, pp. 439–448, 2010.

[95] D. Takahashi, “Computation of the 100 quadrillionth hexadecimal digit of π on a cluster of intel xeon phi processors,” *Parallel computing*, vol. 75, pp. 1–10, 2018.

[96] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster, “Swift/t: Large-scale application composition via distributed-memory dataflow processing,” in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pp. 95–102, IEEE, 2013.

[97] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster, “Swift/t: Scalable data flow programming for many-task applications,” in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 309–310, 2013.

[98] M. Baker, B. Carpenter, G. Fox, S. Hoon Ko, and S. Lim, “mpijava: An object-oriented java interface to mpi,” in *Parallel and Distributed Processing: 11th IPPS/SPDP’99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and*

Distributed Processing San Juan, Puerto Rico, USA, April 12–16, 1999 Proceedings 13, pp. 748–762, Springer, 1999.

[99] B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox, "Mpj: Mpi-like message passing for java," *Concurrency: Practice and Experience*, vol. 12, no. 11, pp. 1019–1038, 2000.

[100] O. Vega-Gisbert, J. E. Roman, and J. M. Squyres, "Design and implementation of java bindings in open mpi," *Parallel Computing*, vol. 59, pp. 1–20, 2016.

[101] M. Nissato, H. Sugiyama, K. Ootsu, T. Ohkawa, and T. Yokota, "Realization and preliminary evaluation of mpi runtime environment on android cluster," in *Advanced Information Networking and Applications: Proceedings of the 33rd International Conference on Advanced Information Networking and Applications (AINA-2019) 33*, pp. 407–418, Springer, 2020.

[102] Z. Yannes, *Portable MPICH2 clusters with Android devices.* PhD thesis, 2015. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2023-03-04.

[103] R. Jain, *The Art of Computer Systems Performance Analysis.* John Wiley and Sons, 4 1991.

[104] R. Buyya, C. Vecchiola, and S. T. Selvi, *Mastering cloud computing: foundations and applications programming.* Newnes, 2013.

[105] M. Poess and R. O. Nambiar, "Energy cost, the key challenge of today's data centers: a power consumption analysis of tpc-c results," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1229–1240, 2008.

[106] M. Dayarathna, Y. Wen, and R. Fan, "Data center energy consumption modeling: A survey," *IEEE Communications surveys & tutorials*, vol. 18, no. 1, pp. 732–794, 2015.

[107] Q. Zhang, Z. Meng, X. Hong, Y. Zhan, J. Liu, J. Dong, T. Bai, J. Niu, and M. J. Deen, "A survey on data center cooling systems: Technology, power consumption modeling and control strategy optimization," *Journal of Systems Architecture*, vol. 119, p. 102253, 2021.

[108] T. Atmaca, T. Begin, A. Brandwajn, and H. Castel-Taleb, "Performance evaluation of cloud computing centers with general arrivals and service," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2341–2348, 2016.