Data Centers from Discarded Cell Phones

Jennifer Switzer, Eric Siu, Subhash Ramesh, Ruohan Hu, and Emanoel Zadorian

***Abstract*—Discarded consumer electronics are a significant and growing source of hazardous waste, with 150 million smartphones discarded each year in the U.S. alone [1]. Repurposing these devices has the potential to extend their lifetime; however, reusing smartphones for general computational tasks is difficult due to the variety and domain-specificity of mobile OS' and hardware. In this we work present a proof-of-concept implementation of a smartphone data center that leverages Raspberry Pi-based cluster management, and a non-traditional mobile OS, to overcome these limitations.**

We evaluate the performance of our implementation in terms of response time, reliability, and energy efficiency. Our experience indicates that reusing smartphones as cloud servers is feasible, and that the resulting system is appropriate for many (but not all) computational workloads.

***Index Terms*—IEEE, IEEEtran, journal, LaTeX, paper, template.**

I. Introduction

ONE hundred and fifty million phones are discarded each year in the US alone, many before they need to be. Despite their nominal 10-year lifespan, most phones are discarded within two years [2]. This is especially problematic because these devices are difficult to recycle; because of their specialized construction and classification as hazardous waste, they cannot be simply thrown into a traditional recycling bin . Even when smartphones make their way to an E-waste recycling facility, these facilities are often unregulated, employing child labor and exposing workers to hazardous chemicals [3], [4].

In this work, we take an alternative approach: Expanding the lifetime of discarded smartphones by reusing them for general computational tasks. Previous work has indicated the feasibility of this approach, but many systems challenges remain. Mobile operating systems, while optimized for their use-case, tend to get in the way of long-term, unsupervised device deployments [5], [6]. For instance, Android includes several battery optimization settings that may kill background processes without warning [7]. Smartphones also tend to boast less computational power than other consumer electronics [8]. And their relatively flimsy constructive (e.g. the inclusion of touch screens) makes them more susceptible to physical damage. We approach this problem as a distributed systems' problem, and implement our smartphone server as a cluster of phones, supervised by a management device that also provides a single point of entry for the user. We replace the Android operating system with Ubuntu Touch, an open-source OS for mobile devices [9]. This allows us to treat the phones as simple Linux machines. To the outside user, our smartphone cluster should appear no different than any other Linux cluster.

II. Related Work

In [6], Shahrad and Wentzlaff propose a server built from decommissioned mobile phones. The proposed server is composed of Samsung Galaxy Note 4's and includes a built in power supply, router, 3 rows of fans, and 84 slots for mobile devices. They propose the use of USB trees for communication between the master node and the phones. A key distinction between their work and ours is that their proposed solution does not include an implementation. Our work is smaller in scale but with an associated implementation, and was inspired in part by a desire to test the real-world feasibility of their approach.

In [10], the authors propose a new, non-traditional cluster computing architecture called **FAWN**, which uses low-powered processors to run massively parallel data IO-based workloads. The primary use-case that the authors were trying to address with this system was to build a much more energy efficient alternative to traditional key-value stores, like **AWS Dynamo DB** and **Memcached**. Since the deployment of traditional, large-scale key-value stores consumes significant amounts of power, the authors propose an alternative cluster design comprised of only low powered processors connected to flash storage, which, as the authors note, is faster and more energy efficient than traditional disk-based stores. Although this work is similar to our project in that it investigates the use of low-powered compute nodes for a cluster setup, it is important to note that a key difference between **FAWN** and our work is that our system focuses more on compute-bound tasks, instead of IO-based tasks. After all, with our project, we aim to thoroughly analyze the feasibility of re-purposing old smartphones as low powered compute nodes for running compute-bound tasks. This is also illustrated via the design of our bench-marking suite (described later in the paper), in which we run many computationally intensive tasks across our system to measure not only the power consumption of our setup, but also its computational efficiency.

An orthogonal line of work is the use of smartphones as edge devices. While this use-case differs from our target, work in this area does highlight the difficulty of using smartphones as unsupervised compute devices. Much of the work in this area (see, for instance, [11]'s air quality monitoring system, and [12]'s activity-level sensor) relies on a human to operate the system and maintain the health of the device. A user is expected to be present to charge the device, respond to software failures (e.g. re-opening the app if it crashes); and protect it from physical damage.

The relative difficulty of deploying smartphones as unsupervised devices is relayed by Klugman et al via their experience deploying a smartphone monitoring system for monitoring the health of power grids in Tanzania [5]. They found that

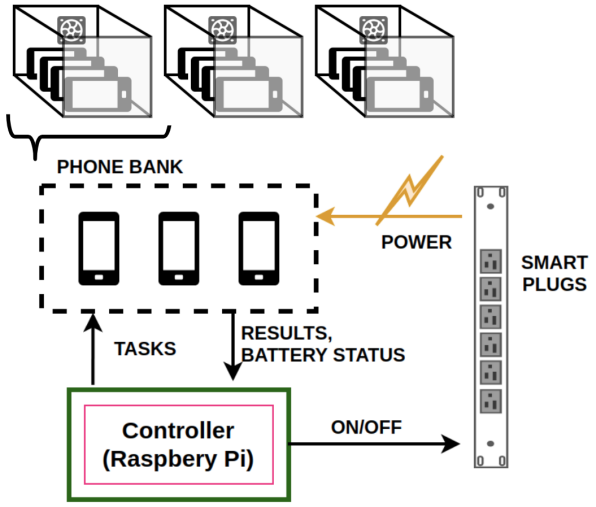

Fig. 1. A diagram showing how the smartphones, Raspberry Pi, and smart plugs are connected within our system

two factors limited the usefulness of the devices: the fact that the Android OS expects human input, and is therefore not well-suited to long term, unmonitored deployments, and the physical degradation of the phones, which experience screen burn-in and battery swell. This experience motivated our decision to replace the Android OS with Ubuntu Touch, and to manage device power via smartplugs.

III. Design Overview

Our cell phone data center consists of three primary components:

1) A central **manager** (Raspberry Pi) that is responsible for managing the devices and distributing tasks among the phones
2) The **phone bank** itself (a collection of used smartphones re-purposed for our project)
3) A **power strip** that provides a power source for each phone via a separate smart-plug, enabling each phone to be toggled on/off remotely

These components and their interactions are summarized in Figure 1. In our current setup, each submitted job includes its public git repo URL and any run-time requirements for the job (ie., amount of CPU/memory required and max running time). Moreover, the git repo for the job must include a **main.sh** file in the root folder, specifying the commands to run for the job. With this job structure, the manager is able to assign each job to the phone that has sufficient resources to run it, and the phones are able to run a job by cloning its git repo and executing its **main.sh** file.

Furthermore, each phone in the phone bank is responsible for running compute jobs assigned to it by the manager, in addition to sending periodic heart-beat messages to the manager so that the Raspberry Pi knows that the phone is still healthy and operational. These heart-beat messages also contain important phone-level metrics (e.g., the current CPU usage and battery power), which enables the manager to keep track of the average historical load of each device (which will be helpful when assigning a new job to a device) and toggle the power to the phone if needed (to maintain the battery levels of all the phones between 20-80%).

The manager is responsible for assigning new jobs to run among the devices, and for dealing with job-level and phone-level failures. When a new job is submitted to our system, the manager first queries for the most healthy device available at that time, which involves filtering all healthy phones that are not currently running any job to pick the one with the lowest average CPU usage, based on the last few heartbeats received from the devices. Once the manager assigns a job to a device, it waits for the device to acknowledge the job, before officially registering the device to that job in its database (which will prevent any future submitted jobs from being assigned to that device while it's still working on this job).

In addition to assigning jobs to devices, the manager also deals with phone-level and job-level failures. The primary type of phone-level failure that the manager handles is when a phone becomes unreachable (i.e. when the phone is no longer sending heart-beats to the manager). The manager periodically queries for phones that have not sent any heart-beats within a reasonable, modifiable time interval, and de-activates them from our system, including toggling their power off and marking them as unhealthy in our database, which prevents future jobs from being assigned to these devices. Moreover, since our data-center is composed of used smart-phones (which are relatively unreliable as compute nodes), every submitted job also has a configured max run-time defined by the end user. The manager uses this property to periodically scan for jobs running past their configured max run-time – at a fixed, modifiable time interval. When this happens, the manager instructs the device to kill the job and re-schedules these jobs (up to a configured max number of retries). This adds a layer of redundancy for the job scheduling process in order to tolerate any intermittent transient failures that may arise during a job's execution.

Finally, we package all of these components into a 3D-printed chassis with built-in cooling to reduce the risk of overheating.

IV. Implementation

We implement our cluster management as a Python-based server that runs on the Raspberry Pi management device, and an associated phone client. This section explains the function of these components, and their interactions.

A. Raspberry Pi Manager

The Raspberry Pi manager starts off by creating a database to keep track of all phones and jobs. Afterwards, the manager runs an Flask HTTP server that is augmented with a SocketIO server. With the HTTP server, the manager receives task requests from the end user via POST requests, where each task contains a URL link to a GitHub repository as explained in Section III. Upon receiving the job, the manager creates a Job JSON object within the database to keep track of its

Interface	SocketIO Event / HTTP Request	Description
/devices/register	HTTP POST	Registers a phone with the manager
/devices/<int:device_id>/heartbeat	HTTP POST	Sends a heartbeat update to the manager
/devices	HTTP GET	Obtains a list of all devices
/jobs	HTTP GET	Obtains a list of all jobs
/jobs/submit	HTTP POST	User end-point to send a job
/jobs/<int:job_id>/status/	HTTP GET	Returns the status of a job
/jobs/<int:job_id>/update_status/	HTTP POST	Updates the status of a job
/devices/register	HTTP POST	Registers a phone with the manager
cancel_job	SocketIO Event	Manager-emitted event to tell a phone to cancel a job
task_acknowledgement	SocketIO Event	Phone-emitted event to tell manager that a job has been acknowledged

TABLE I
MANAGER API SUMMARY

status, GitHub URL, and resource requirements. If there are phones available, then we can schedule that job to a phone through the SocketIO server by emitting a message with the GitHub URL and the device id of the phone. A summary of all HTTP and SocketIO interfaces is shown in Table I. The manager also toggles charging for all the phones via IFTTT (If This Then That) on the Wyze smart plug that the phone is connected to via a webhook secret key.

We determine the availability of a phone based on many metrics: CPU usage, active status, and cooperativeness. The CPU usage is used to choose a phone who is being used the least and will be able to handle the workload, however, if all phones are too busy, we send back a "NO_DEVICES_ARE_AVAILABLE" HTTP 500 error code, but also schedule the job for later in case a phone does open up. The active status of the phone is determined by asking if the phone is able to send heartbeat messages through HTTP within a minute of their past heartbeat message. If the phone fails to send a heartbeat within each minute, then we consider that phone as inactive and unable to be assigned jobs, wherein their charging port is turned off. Lastly, cooperativeness (or rather the lack of) is determine by the number of times a phone has failed to acknowledge working on a job and failed to finish a job. A phone has up to 10 seconds to send an acknowledgement of receiving a job to the controller and up to the max_runtime_secs given by the end user to finish a task. A failure on either of these will increment their summed failure counter, where upon 100 failed attempts will automatically decommission the phone for being uncooperative.

The manager is able to determine the failure of these phones by periodically checking the phones and the jobs. For each minute, the manager checks the phones to see if they have failed at least 100 times, if so then they are decommissioned and removed from the power strip until a human operator can replace the phone or recommission them. For every 5 seconds, the manager checks the jobs to see if any jobs timed out or failed to be acknowledged. In a failed instance, jobs are retried/rescheduled until they've reached their MAX_JOB_ATTEMPTS, which is 5.

For phones that didn't fail, they successfully send a task_acknowledgement Socketio event back to the manager, which removes their pending acknowledgement from a list. One may be confused as to why we have a http server and a SocketIO server. The SocketIO server allows the phone to connect to the manager without the manager having to remember the IP addresses of all the clients. If we were only using HTTP, then we would have to manually send a request to the phones every time we required them to perform a task. Having a mapping of IP address to a device id is a level of indirection that is easily removed by using SocketIO. By sending the device id over the emit message, the phones will be able to tell which job is assigned to which phone without the manager having to try to determine which phone belongs to which IP address.

B. Phone Client

Each phone client runs two scripts: one for heartbeat messages and one for the SocketIO interface. The phone client must first send a HTTP POST request to the manager's "/devices/register" route with their device id and smartplug webhook key. After registering the device, the phone client can connect to port 5000 of the manager's IP address to connect to the SocketIO server using the "client.py" file. After connecting to the manager, the phone can receive SocketIO messages on task_submission events that will contain the GitHub URL for the pending task. After receiving the job, the phone will send a task_acknowledgement to the manager before starting the job. The phone will run the main.sh script to begin the job. After finishing the job, the phone will either set the job status with STATUS_SUCCEEDED or STATUS_FAILED with the server's /jobs/<int:job_id>/update_status/ API depending on the output of the job.

As for the heartbeat messages, the phone runs a "send_heartbeat.py" script that repeatedly sends the phone's CPU usage and battery levels every 5 seconds to the server route of "/devices/<device_id>/heartbeat". From this heartbeat, the manager will be able to control the charging on the phone. The phone will start charging when its battery is lower than 20% and stop charging when it hits at least 80%. If these heartbeat messages were to stop for at least one minute, the phone will be treated as inactive and will not have jobs assigned to them.

C. 3D-Printed Chassis

We have a 3D printed white chassis that has glass on its sides and two fans on each end of the case, this is shown in Figure 2. We have holes on the top of the phone bank to plug chargers into the phone. We have placed the phones upright to allow for maximal air flow for all the phones with the two installed fans.

Fig. 2. Left: Inside View, Middle: Top View, Right: Front-Side View

	Total Response Time	Useful Compute Time	Total Overhead
1 phone, 1 job	56.1s	43.0s	13.1s
Fail/reschedule	131.6s	43.0s	88.6s
No phone, 1 job	0.56s	N/A	0.56s

TABLE II
RESPONSE TIME UNDER THREE DIFFERENT CIRCUMSTANCES.

V. EVALUATION

A. Performance

1) Response Time & Reliability.: We define the response time of the datacenter to be the time elapsed between a job submission to the Raspberry Pi, and the return of a completed result or failure message to the client via the Pi. We measured this for several scenarios, each of which is described below. A summary of the results is presented in Table II.

- **Single-phone, single-job:** A single phone is connected, and the client submits a single job. No large-scale failures occur.
- **No phones, job submitted.** A job is submitted, but no phones are connected. An error message is returned to the user.
- **Job submitted & rescheduled.** Two phones are connected. The client submits a single job, and the job is allocated to one of the phones (Phone A). Phone A is disconnected from the network, and the job is re-allocated to the second phone (Phone B).

The time needed to reschedule a job is dictated by time interval based parameters: job check frequency, heartbeat timeout duration, and a job's user-defined run time. For job check frequency, we ensure that any unacknowledged jobs and timed-out jobs are rescheduled by checking all the jobs, then waiting a server-defined time interval before checking again. For heartbeat timeout duration, the controller will ensure that phones are active by checking if they have sent a heartbeat message within a server-defined time duration, otherwise their jobs will be rescheduled and new jobs will not be assigned to the device. If the phones have acknowledged the jobs and are active, then rescheduling depends on the job's user-defined time, which defines when the checker should tell the phone to time-out and reschedule the job.

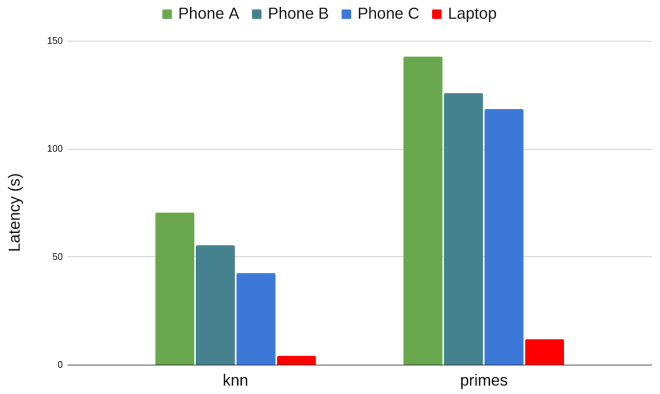

Fig. 3. The phones (shown in green, teal, and blue) are approximately 30x slower than a consumer-grade laptop (shown in red).

B. CPU Benchmarking

Development phones. We measure the performance of our three Nexus 4 development phones, as compared to a consumer laptop.

We test the performance of the phones on two of our example jobs, training a knn classifier and identifying large prime numbers against a consumer-grade laptop [1]. Although there is variation among the phones, they are all approximately 30x slower than the laptop.

Android in general. In addition to benchmarking our prototype phones, we also performed CPU Microbenchmarking on a variety of other used Android phones. All of these phones were in use for at least a year before being donated to us. These benchmarks were performed using the Geekbench 5 app, a CPU-benchmarking app that is available on the Google Play store [8].

Our results (Figure 4) indicate that smartphone CPU power is increasing quickly with time. Our Samsung Galaxy S10 (released 2019) has a performance similar to that of Intel's Core i3-8100 processor (released 2017).

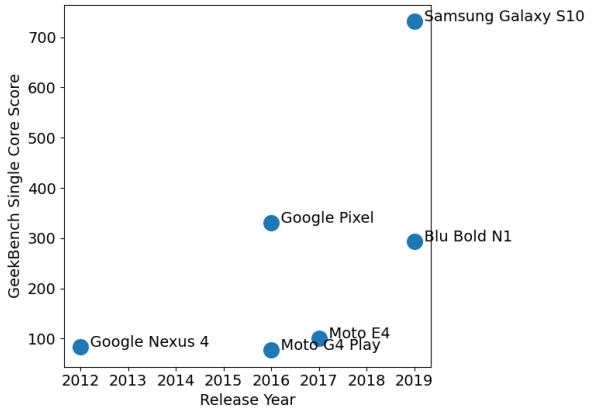

Fig. 4. Geekbench results for several used Android phones. The GeekBench metric is presented out of 1000, with a score of 1000 being equivalent to the performance of a an Intel Core i3-8100.

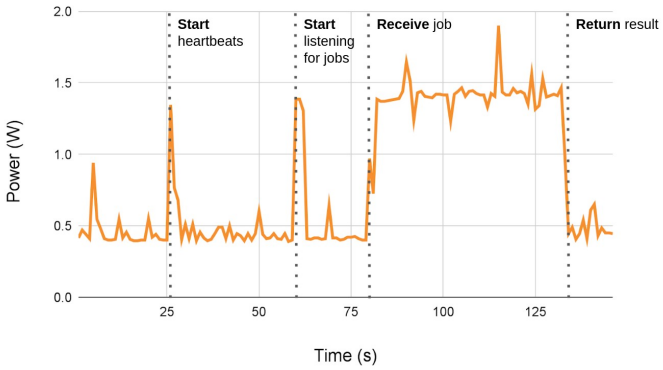

Fig. 5. Power draw over time, with events marked at the dotted lines. This data was gathered via a USB power meter for a phone at full charge plugged into power.

C. Energy Consumption

We measure the energy consumption of the one of our development phones for a full cycle of start-up, registration, job acceptance, and job completion. The results are shown in Figure 5. The phone draws approx. 0.5W at rest, and 1.5W while completing a job. If we assume that we run our data center at 95% capacity (e.g. each phone is occupied 95% of the time), this would imply a daily energy consumption of 125.3kJ per phone. Since this is almost 5x the battery capacity of a new Nexus 4 phone, it seems likely that the phones would have to be plugged into power almost constantly to run at this capacity.

VI. Conclusion

We have explored the feasibility of using discarded smartphones as general-purpose compute nodes. Overall, our experience shows that even 8 year old phones can support common computing tasks, and can be combined into general-purpose compute clusters. This strategy has the potential to reduce harmful e-waste, and provide an affordable cloud-computing alternative.As newer phones are released, and newer phones are thrown out, the performance of the system will only improve.

[1]A 3rd gen Lenovo Thinkpad X1 Carbon with 8GB of system memory and an Intel(R) Core(TM) i7-5600U CPU @ 2.60GHz

Acknowledgment

The authors would like to thank Professor Ryan Kastner for his invaluable feedback, and Narek Boghozian for his help with the case.

References

[1] 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)*, 2014, pp. 57–62.

[2] R. Geyer and V. D. Blass, "The economics of cell phone reuse and recycling," *The International Journal of Advanced Manufacturing Technology*, vol. 47, no. 5-8, pp. 515–525, 2010.

[3] C. P. Baldé, V. Forti, V. Gray, R. Kuehr, and P. Stegmann, *The global e-waste monitor 2017: Quantities, flows and resources*. United Nations University, International Telecommunication Union, and . . . , 2017.

[4] 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. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0956053X18306366

[5] 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*, ser. MobiCom '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 849–851. [Online]. Available: https://doi.org/10.1145/3241539.3267726

[6] M. Shahrad and D. Wentzlaff, "Towards deploying decommissioned mobile devices as cheap energy-efficient compute nodes," in *Proceedings of the 9th USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'17. USA: USENIX Association, 2017, p. 6.

[7] "Background optimizations," https://developer.android.com/topic/performance/background-optimization.

[8] "Android benchmarks - geekbench," https://browser.geekbench.com/android-benchmarks.

[9] "Ubuntu touch," https://ubuntu-touch.io/, accessed: 2021-04-12.

[10] 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, no. 7, p. 101–109, Jul. 2011. [Online]. Available: https://doi.org/10.1145/1965724.1965747

[11] D. Hasenfratz, O. Saukh, S. Sturzenegger, and L. Thiele, "Participatory air pollution monitoring using smartphones," *Mobile Sensing*, vol. 1, pp. 1–5, 2012.

[12] M. Keally, G. Zhou, G. Xing, J. Wu, and A. Pyles, "Pbn: Towards practical activity recognition using smartphone-based body sensor networks," in *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 246–259. [Online]. Available: https://doi.org/10.1145/2070942.2070968