DAMMP: A Distributed Actor Model for Mobile Platforms

Arghya Chatterjee*
School of Computer Science,
Georgia Institute of Technology
Atlanta, Georgia, USA
arghya@gatech.edu

Srđan Milaković
Department of Computer Science,
Rice University
Houston, Texas, USA
sm108@rice.edu

Bing Xue
Department of Computer Science,
Rice University
Houston, Texas, USA
bx3@rice.edu

Zoran Budimlić
Department of Computer Science,
Rice University
Houston, Texas, USA
zoran@rice.edu

Vivek Sarkar
School of Computer Science,
Georgia Institute of Technology
Atlanta, Georgia, USA
vsarkar@gatech.edu

ABSTRACT

While mobile computing has seen a trend towards miniaturization and energy savings for a number of years, the available hardware parallelism in mobile devices has at the same time continued to increase. Overall, mobile devices remain resource constrained on *power consumption* and *thermal dissipation*. Aggregating the computing capabilities of multiple mobile devices in a distributed and dynamic setting, opens the possibilities for performance improvements, longer aggregate battery life and novel dynamic and distributed applications.

In this paper, we propose a Distributed Actor Model for Mobile Platforms (DAMMP), which includes *a*) a mobile extension to the actor-based *Distributed Selector (DS)* programming model, along with a new implementation for mobile Android devices, *b*) an extension to the DS programming model that enables the programmer to react and adapt to dynamic changes in device availability, *c*) an adaptive mobile-to-server and mobile-to-mobile computation offloading model and its implementation on the Android platform, and *d*) creation of a dynamic network of heterogeneous Android devices using both Wi-Fi Soft AP and Wi-Fi Direct's peer to peer (P2P) network.

We evaluate the DAMMP framework under ideal thermally-controlled usage conditions to show promising scalability and performance, and analyze the communication overhead of both Wi-Fi and Wi-Fi Direct when used as the communication layer for DAMMP. We also evaluate the impact of adaptive offload on device-level thermal dissipation in more realistic usage scenarios, thereby demonstrating possibilities for thermal control and power management that can be achieved at the application level with a distributed actor model. To the best of our knowledge, this work is the first cross-platform distributed actor/selector runtime system that can span mobile devices and distributed servers.

*Also with Computer Science and Mathematics Division, ORNL.

ManLang 2017, Prague, Czech Republic

DOI: 10.1145/3132190.3132205

CCS CONCEPTS

• **Computing methodologies** → **Parallel programming languages**; *Distributed programming languages*;

KEYWORDS

Actor Model, Selector Model, Mobile Computing, Android Platform, Distributed Selectors, Remote Messaging, Remote Synchronization.

1 INTRODUCTION

The computational demands on mobile devices are increasing at a rapid pace that is hard to sustain with current device-level energy capacities. In this paper, we make a case for using a Distributed Actor Model for Mobile Platforms (DAMMP) to harness the processing power of multiple supporting mobile devices, and (when available) supporting servers, so as to increase the responsiveness and reduce the energy consumption of specific mobile devices that need to process compute-intensive jobs.

While mobile devices are not usually associated with distributed and high-performance computing, past projects including distributed peer-to-peer file sharing for mobile applications [20] and off-the-grid agricultural distributed mobile applications [21], have demonstrated real-world impact with distributed mobile applications. However, while most such projects are based on specialized hardware, we aim to provide a *portable* and more *extensible* framework with DAMMP. Some popular scenarios from past work on image recognition [9, 24] and GPS triangulation [17, 27, 32, 33] help motivate our model in dynamic and heterogeneous settings:

- Law enforcement agencies in the field may not have access to high-performance computing resources to run a facial-recognition application, and may have to rely on the computing power of their hand-held mobile devices instead. These compute intensive jobs may not be possible to run on a single device, but with the use of DAMMP they can transparently create a distributed network of multiple devices to collaboratively implement the face-recognition algorithm [9].
- Hikers are at a remote location without Internet connectivity and with a set of devices with different computing power, battery states and temperatures. A DAMMP application can enable them to share GPS data among their devices and triangulate their position using the collective

computing power of their devices, while at the same time managing the power consumption and heat dissipation to maximize the collective usability and longevity of their ad-hoc distributed system of heterogeneous devices [17].

Scenarios such as these illustrate a need for a programming model and system that would allow an easy distribution and redistribution of work that needs to be done, a convenient mechanism for the application to adapt to dynamic changes in network topology (such as devices leaving or joining the group), and an effortless model for offloading computation to another device as a reaction to dynamic environment changes (such as battery running low or temperature running high on a device). We believe that the DAMMP programming model and runtime is well suited to addressing this need. DAMMP combines the distributed asynchrony inherent in the *Actor model* (which greatly facilitates migration), the expressiveness of the *Selector model* (which can express a wider range of coordination protocols), distributed execution on ad-hoc networks using *Wi-Fi Direct*, and an Actor-friendly interface between the runtime and the application that allows for an easy implementation of adaptation mechanisms that react to dynamic changes in the system (devices joining/leaving, battery running low, temperature rising too high, OS throttling the CPU, etc.).

To the best of our knowledge, this work is the first cross-platform distributed actor/selector runtime system that can span mobile devices and distributed servers. The specific contributions of this paper are as follows:

(1) We present a Distributed Actor Model for Mobile Platforms (DAMMP), a Java library that extends the Distributed Selector model [8] to include support for mobile platforms as well as support for dynamic topology changes and reconfiguration.
(2) DAMMP includes a clean and intuitive interface for the user to implement reactions to changes in the system topology and configuration.
(3) An implementation of this programming model on modern Android devices.
(4) We introduce an adaptive model for offloading computations from a mobile device on to nearby mobile devices, and (when available) nearby servers.
(5) We demonstrate and evaluate the programming model based on a range of use cases and metrics (performance and energy) for heterogeneous Android devices.

The rest of the paper is organized as follows. Section 2 summarizes background on the general Selector Model, the Distributed Selector (DS) model and Wifi-Direct, Section 3 describes the technical approaches that we used to adapt and extend a Distributed Actor runtime for Mobile Platforms, followed by the evaluation of our Android implementation of DAMMP in Section 4. Section 5 summarizes related work in distributed computing with the use of the Actor programming model. Finally, in Section 6 we briefly conclude with some possible directions for future work.

2 BACKGROUND

Throughout this paper, we will use the term, *Selector*, for entities defined by DAMMP users, and the term, *Actor*, for entities internal to the DAMMP runtime (which happen not to require the multiple-mailbox functionality available in selectors).

2.1 Selector Model

The Selector model [14] is an extension of the Actor programming model [1] that supports multiple guarded mailboxes (which can be enabled or disabled independently) along with the priority-based processing of messages. It overcomes known difficulties in implementing synchronization and coordination patterns using the pure actor model. The idea of guarded mailboxes in selectors is inspired by classical condition variables, in which a thread checks whether a condition is true before continuing execution in a critical section. The results in [14] show that Selectors can also be implemented efficiently, since that work includes performance comparisons with Scala, Akka, Jetlang, Scalaz, Functional-Java and Habanero actor libraries. However, the implementation described in [14] was limited to a single-node (shared-memory) implementation of the Selector model in the Habanero Java Library [12].

2.2 Distributed Selector Model

The Habanero Java Distributed Selector (HJDS) runtime library [8] extended the design and implementation of the single-node Selector library from [14] to work across multiple distributed server nodes (with one or more JVM instances per node). HJDS makes it possible to use selectors as a single unified programming model for both shared-memory and distributed multi-node execution of a program.

The HJDS runtime allows the programmer to focus on implementing algorithms for solving a problem without worrying about whether the application should run on a shared-memory or distributed-memory system. The runtime also provides automated system bootstrap and distributed global termination by terminating the entire distributed system when all selectors in the user code have been successfully terminated.

For a single HJDS program, each computing node is set up with a configuration file which can be customized by the user to adjust the number of computing nodes, and JVM processes per node, used by the program. The master node sets the program executables on all remote nodes and starts up a process on each node to initiate distributed program execution. The global termination is initiated by the master node and is performed in stages to detect when the user program becomes quiescent.

The contributions of [8] include *a*) transparent creation of selectors on remote nodes, and *b*) seamless message delivery system for both local and remote nodes, while keeping the functional differences between local/remote communication transparent to the programmer. With the high-level abstraction and location transparency of the programming model, the HJDS runtime library demonstrates high programming productivity for distributed applications.

2.3 Wireless communication

With the IEEE 802.11 standard becoming one of the most successful wireless protocols for accessing the Internet, Wi-Fi technology has also been extended to accommodate P2P device connections beyond the traditional approach of using Access Points (APs). Specifically, the *Wi-Fi Direct* technology is developed by the Wi-Fi Alliance to expand the use cases of Wi-Fi technology for device-to-device

communication [6]. It builds upon the IEEE 802.11 infrastructure mode and uses devices as logical SoftAP (Software-enabled Access Point) for connectivity, without relying on external AP support as in ad-hoc networking. Device-to-device communication in a typical Wi-Fi network has to be supported by the external APs. However, in a Wi-Fi Direct P2P network, the logical role of AP can be specified dynamically, and can exist on any client device. Devices with Wi-Fi Direct capabilities can communicate by forming P2P groups.

The group formation process has several phases. Before group establishment, devices are in a *discovery* phase, which is performed by a traditional Wi-Fi scan. A device can either discover an existing P2P Group, or some devices can discover each other. When a device discovers an existing P2P group, it may choose to query the set of current services on the group and join based on the information. A device that did not discover any existing P2P group, or other devices to form a group can *autonomously* create a group and become the Group Owner (GO). When devices discover each other, they may enter a *negotiation* phase to determine which device would be the Group Owner (GO) and serve as a Soft-AP [15]. Once the GO role is established, clients can choose to join the P2P group through the discovery of the GO.

Compared with traditional device-to-device connectivity technologies such as Bluetooth and ZigBee [26, 36], with nominal ranges from 10 meters to 100 meters, and transfer speed between 250 kbps to 25 Mbps, Wi-Fi Direct, inherits all the capabilities from IEEE 802.11 standards, and claims to provide a nominal range up to *200 meters* and transfer speed up to *250 Mbps* [19]. Power saving support and extended QoS capabilities; Wi-Fi Direct can be considered one of the most promising candidates for wide range device-to-device communication, and suitable for our goal of distribution across mobile devices.

Android 4.0 and later versions comply with the Wi-Fi Direct certification program and allows applications to interact with other devices without using an external network connection [3]. The Android Wi-Fi Direct interface (`WifiP2pManager`) allows developers to discover, request and connect to peers, and provides listener methods that detect the success or failure of connected/dropped connections and newly discovered peers. The Android API does not implement any particular GO negotiation algorithm, and each client can only belong to one P2P group at a given time.

After the maturation of client-server based wireless communications, energy efficient and high bandwidth direct device-to-device communication, will be the new challenge faced by mobile platforms in creating more secure and more accessible distributed applications and systems. Since Wi-Fi Direct technology is not as yet widely available, in this work, we also consider Wi-Fi Hotspot (Soft AP) on Android as a proxy due to its commonly available hardware support. The use of Wi-Fi direct can significantly reduce the network level complexity that needs to be handled by the end-user, such as discovering new devices in the network, connecting these devices and even notifying the connected devices when any particular device drops from the network.

3 TECHNICAL APPROACH

In this section, we discuss the technical contributions of this paper. First, we describe the extensions made to the Distributed Selector model to support mobile platforms [Section 3.1]. Next, we describe how our approach can enable user-level adaptation based on changes in system topologies and configurations [Section 3.2]. Then, we discuss how our approach can enable resilient and adaptive computation offloading with heterogeneous devices [Section 3.3]. Finally, we summarize how our approach can enable a unified distributed computing model across mobile devices and distributed servers [Section 3.4].

3.1 Extension of Distributed Selector model for mobile platforms

Our work extends the HJDS execution model [8] to mobile platforms. There are several challenges that had to be addressed in this extension including dealing with unpredictable *periods of unavailability* for different devices, and ensuring that a *reliable runtime system* can be implemented across mobile devices. Our mobile implementation fully supports the lightweight location-transparent actor creation and actor-based runtime control mechanisms described in [8]. In this work, we limit a single physical device to host exactly one *place* (the logical unit for a shared-memory runtime instance that can host multiple actors). Our approach extends HJDS's non-resilient high-performance runtime into more decentralized mobile platforms through the following innovations:

- We introduce a fully decoupled communication middleware layer for the mobile platform,
- We can optionally expose and delegate some of the runtime control to the application level, and, finally,
- In order to support the distributed actor runtime on *volatile* mobile networks, we have replaced the automated bootstrap and global termination feature in HJDS that assume stable network connectivity by a new communication manager that is more suitable for mobile computing.

The cluster-based implementation of HJDS assumes network stability and low communication costs. However, those assumptions usually do not hold on volatile mobile platforms such as the one we are considering in this paper. Over-the-air data transmission is a lot more expensive relative to that of wired connections, and the volatile nature of mobile networks due to devices joining, leaving, dropping out, going out of range or running out of power makes the automated bootstrap and global termination features in HJDS to be of limited use for the mobile platforms that we are considering. In addition, automated bootstrap and global termination require a monolithic application model that limits the expressibility of the Actor model in many peer-to-peer applications. Instead, our mobile-oriented approach encourages a *decentralized* programming pattern that we further enhance by delegation of limited runtime control to applications on single devices.

An overview of a single place (device) in the selector system is shown in Figure 1. One may consider each device to be a local selector system. A local selector system in a single place (or device) consists of 1) a *runtime system* with a System Actor (SA), a Proxy Actor (PA) and Mobile Communication manager, and, 2) a *developer*

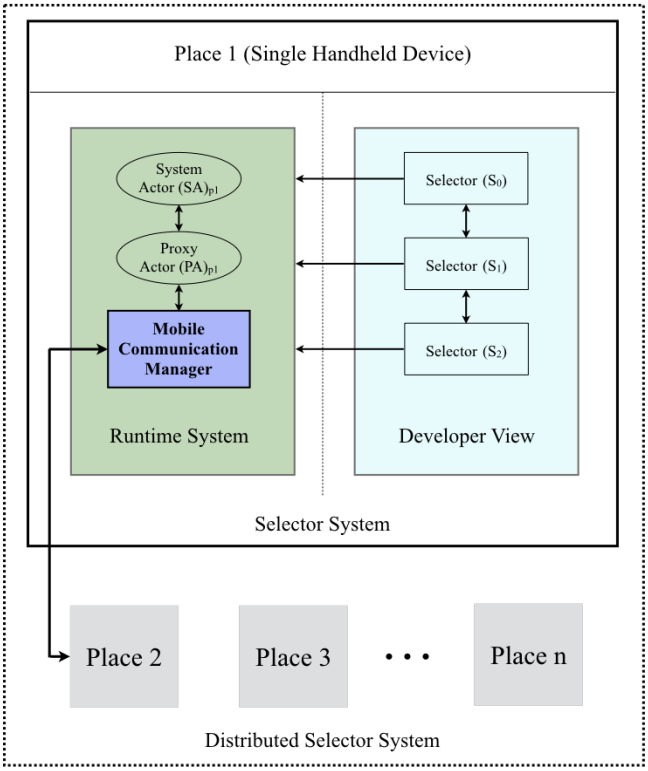

Figure 1: Overview of the distributed selector system. On each hand-held device, we have a single selector system with the Mobile Communication Manager of each device exposed to the entire distributed network. All communications to external handheld devices are performed by the Mobile Communication Manager.

view with one or more user-defined selectors. As shown in Figure 1, the user-defined selectors can only communicate with each other and the runtime system of the place that hosts the selectors. All communications to other handheld devices (places) must be managed by the Mobile Communication manager.

We introduce the `IMobileCommunicationManager` interface (see Listing 1), fully decoupled from the HJDS runtime using a runtime-userspace callback system, based on the Actor model. The `IMobileCommunicationManager` includes a system callback handle that is invoked after any change in network status. Upon creation of a selector system instance, the selector system will call `start()` to initiate the communication manager, and the communication manager will call `ISystemCallback.onConnectionReady` once the device is ready to join a network.

```
1    public interface IMobileCommunicationManager {
2        interface ISystemCallback {
3            void onConnectionReady(Place localNode);
4            void onMessage(Message message);
5            void onPlaceJoin(Place place);
6            void onPlaceLeft(Place place);
7        }
8        void start();
9        void stop();
10       boolean send(Place place, Message message);
11       void setSystemCallback(ISystemCallback callback);
12   }
```

Listing 1: The communication API for mobile platform.

When a connected device leaves the network, the communication manager invokes `ISystemCallback.onPlaceLeft()` to notify the selector system, which in turn will alert the application of a network change via a system message. While there is a callback available for any place joining the network, the selector system can ignore network topology changes due to new devices joining until it is present with a remote selector reference. In other words, new devices joining the network does not affect the selector system until an Actor message is exchanged. The `onPlaceJoined` callback only gets invoked if the application desires to be notified of such topological changes. On the other hand, a device leaving the network can cause a known remote selector reference to be invalid and is immediately reported to the application through the callback system. We guarantee *non-blocking* data transmission in a selector system instance by forcing the communication manager to run on separate threads.

Communication between two handheld devices. Figure 2 shows a message-sending protocol for a general Ping-Pong application using two devices (*place*). Let's consider the first device (device 0) to be the Ping-Place and the other device (device 1) to be the Pong-Place.

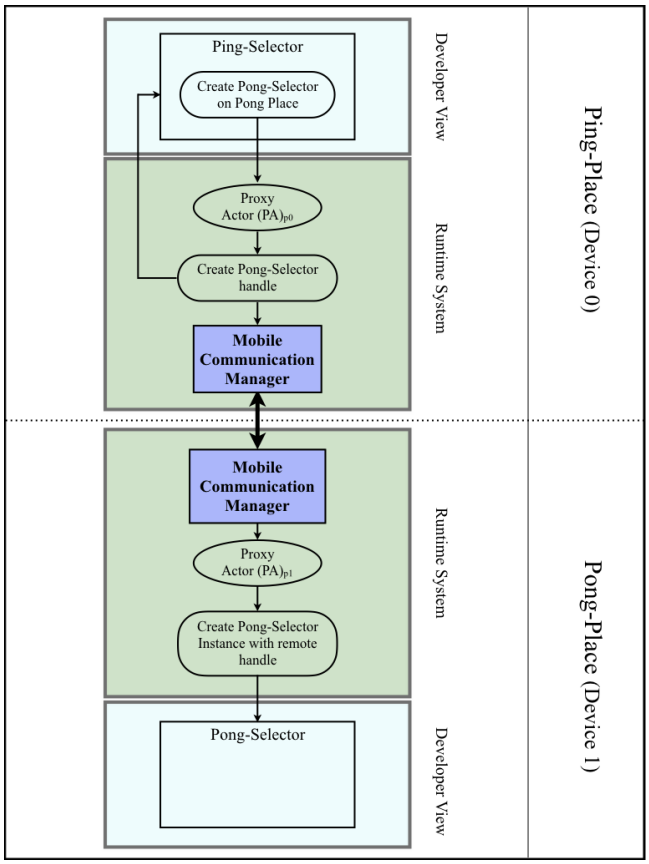

Figure 2: Ping-Pong message sending protocol using the communication manager.

In this scenario, the user-defined Ping-Selector wishes to create a Pong-Selector on the Pong-Place. To that end, the Proxy Actor at

device 0 creates a remote handle for the Ping-Selector to communicate with the Pong-Selector, and the Communication Manager at *device 0* communicates with the Communication Manager at *device 1* to create a Pong-Selector at the Pong-Place. The Proxy Actor at Pong-Place then creates the user-defined Pong-Selector. All further ping-pong messages are handled by the Communication Manager using the remote selector handle.

3.2 User-level adaptation based on changes in System Topologies and Configurations

Past implementations of distributed actors and distributed selectors targeted cloud-like distributed servers, and did not account for the fact that devices can join and leave a mobile network. Due to this volatile nature of mobile platforms, we have extended our distributed implementation with a publish-subscribe model that enables user-level control of adaptation to network changes by application-level actors. As shown in Listing 2, applications can subscribe to specific message types (including runtime-generated alerts) through designated mailboxes.

```
@Subscription(topics = {
   @Topic(messageClass = NodeJoined.class, mailbox = 0),
   @Topic(messageClass = NodeLeft.class, mailbox = 0),
   })
```

Listing 2: A selector class can subscribe to different alerts from the runtime system.

Applications can choose to react to different categories of runtime and communication events in different ways. By assigning mailbox priorities, developers can implement application-specific resilience models without changing the underlying actor-based program semantics.

As an example, Figure 3 shows a dynamic join protocol for adding new devices to a DAMMP applications. When a new device wishes to connect and join the network of devices, it first connects to the Group Owner (GO). Upon successful connection with the GO, the GO sends information about all the other devices that are already connected to the network. The new device can then connect to the other devices in the network as needed.

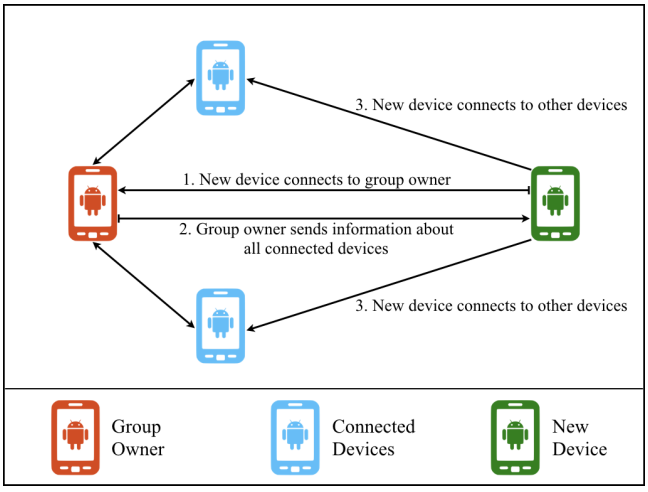

Figure 3: Dynamic join protocol for new handheld devices.

As an example of adaptation, the application can designate different priority values to different system alerts, e.g., by choosing to give higher priorities to topological changes for handling partial failures, and thereby avoid waiting for work-related messages to complete before topological changes are handled. The subscription mechanism also allows the communication manager to communicate directly with the application if a specific implementation calls for user input/interaction. Finally, the communication manager can send custom messages through the selector's system callback and have the messages delivered to any selector subscribed to the custom message class.

Resilience. Our runtime supports a resilient master-worker pattern by periodically backing up program snapshots of the master selector (a *selector* instantiated locally without a parent) on other nodes in the network. Upon a network change, selectors can continue execution as normal as the runtime will buffer all outgoing messages and restore a master copy once the network is re-established. The runtime will halt selector message processing by disabling any non-subscription mailbox once the out-going buffer is full, to prevent any further outgoing application-level message being generated, and alert the application through subscription mailboxes. Upon network re-establishment, if a master selector copy is present in the network, the runtime will restore master selector processing; otherwise, the runtime notifies the application-level and restarts with a fresh copy of the master selector.

3.3 Adaptive offloading across mobile devices and servers

The master-worker pattern is a common building block for parallel and distributed applications, and multiple master-worker patterns can be composed hierarchically when there's a danger of a single master actor becoming a bottleneck.

Typically, the main computation in the master will generate multiple sub-problems that can be delegated to multiple workers for parallel processing, for example in divide-and-conquer parallel algorithms. However, an application written using the DAMMP framework can easily offload sub-problems to other devices. A master-worker based paradigm should allow easy computation offloading through creating workers on more powerful devices (relative to smartphones), such as tablets, laptops and desktops (which can play a "server" role relative to smaller mobile devices).

Our runtime library currently supports five types of message classes, created at the library level (apart from the node joining and node leaving system messages, discussed in Section 3.2) which can be used by the user / runtime to automatically offload certain computations based on a threshold. The message classes are as follows:

- **Battery Status Message:** Battery low / Battery okay
- **Charging Status Message:** Device is connected: AC or USB / Disconnected: AC or USB
- **Battery Level Message:** Battery percentage when battery level changes
- **Temperature Message:** Average temperature in last *'s'* seconds has changed by *'d'* degrees in °C
- **Wifi Signal Strength:** Average Wi-Fi signal strength (1 to 10) in last *'s'* seconds changed by *'w'*

By subscribing to one or more of the above-mentioned message classes, the user / runtime can receive relevant information about the device and the network, and aid in the offloading of the partial computation. Information about the offloading device, can also be obtained by subscribing to the offloading device's message classes. All messages are sent to the Proxy Actor of that device. To mitigate additional overhead due to offloading partial computation, we must observe certain application / network conditions, such as:

- **Network bandwidth:** Using the *Wi-Fi Signal Strength* messages, the user / runtime can get information about the network and can decide to offload right away or wait for better network bandwidth.
- **Application:** The degree to which an application can benefit from offloading depends in part on how much communication is needed between the mobile application and the offloaded computation. In the ideal case, only two messages are required, one for offloading and one for the return value.
- **Offloading Device:** Choosing the correct device to offload computation on is also important in order to achieve performance without over-burdening the offloading device. When a new device is connected to the group, our runtime notifies subscribing applications about the type of the device (phone / tablet / laptop / desktop) that joined the group. Using the device-type information and by subscribing to the *Battery Level Message* and the *Temperature Message* classes of the connected devices, the user / runtime can choose a suitable device for offloading the computation.

3.4 Unified distributed computing model for mobile devices and distributed servers

Using the DAMMP runtime, application developers can tap into the processing power of other devices (e.g., tablets, servers, etc.) by creating a network of heterogeneous devices. The DAMMP framework extends the Actor and Selector programming models that have built-in message-passing semantics, by allowing for natural implementations of distributed actor-based applications on networks of heterogeneous devices. It is possible to deploy the same application on both the cluster-based HJDS runtime, and our DAMMP framework with a Wi-Fi based communication layer. Even with mobile platforms getting faster everyday, there is a strong motivation to enable them to offload computations on to local servers (such as "fog servers" in [5]). For example, hand-held devices such as mobile phones and tablets might be the only computing power that is available within a group of people in some situations. Or, as illustrated in Section 4.2.6, some mobile hand-held devices might lack the sustained energy and computing power to complete a task, and be motivated to use DAMMP to create a heterogeneous mobile network which could include more powerful devices such as laptops and tablets. Applications can readily harvest these computing resources to aid application demands that exceed the capability of on-device chips while also reducing battery consumption and addressing thermal constraints at the same time.

4 EVALUATIONS

4.1 Experimental Methodology

4.1.1 ***Android Implementation.*** Our Android-based implementation of DAMMP currently supports two communication layers, one with standard Wi-Fi and the other with Wi-Fi Direct. We are using the Android operating system as our research vehicle because of its open source software stack and Linux kernel roots, as well as the availability of the Android JVM — Android RunTime (ART) [2], an efficient and low memory footprint virtual machine that provides a high-level managed runtime that is well suited for Actor implementations.

Our implementation was undertaken on Android 5.1.1 and complies with available Wi-Fi and Wi-Fi Direct APIs at level 22. Under the Wi-Fi based communication layer, devices connect to each other through an external hardware access point. Under the Wi-Fi Direct based communication layer, one device acts as Group Owner and broadcasts provided service(s), while nearby devices may join through service discovery to act as Group Member(s). Due to current limitations in the Wi-Fi Direct implementation for Android, there is only one Group Owner per communication group. Developers may also provide application and/or hardware specific implementations of the `IMobileCommunicationManager` interface discussed in Section 3.2.

4.1.2 ***Hardware Setup.*** Our tests platform includes five Nexus 5 devices, with a Quad-core 2260 MHz Krait 400 processor and a Qualcomm Snapdragon 800 MSM8974 system chip, and three Nexus 4 devices, with a Quad-core 1500 MHz Krait processor and a Qualcomm Snapdragon S4 Pro APQ8064 system chip. We used a 2010 MacBook Pro, with a 2.66 GHz Intel Core i7 processor and 8 GB DDR3 memory, as the target for offloading computations from the Nexus 4 and Nexus 5 devices.

4.2 Benchmarks

We provide experimental results and analysis for three sets of benchmarks:

(1) Two distributed actor benchmarks that measure the scalability of our distributed mobile platform (Sections 4.2.1 and 4.2.2)
(2) Two micro-benchmarks that measure message passing throughput, and the impact of communication overhead on application scalability, in different communication environments (Sections 4.2.3 and 4.2.4)
(3) A distributed actor benchmark that shows the usability of our adaptive offloading model (Section 4.2.5)
(4) Two distributed actor benchmarks that measure the impact of computation offloading on scalability and thermal dissipation (Section 4.2.6)

Since these are standard actor benchmarks that do not need multiple benchmarks, they are all implemented as selectors with a single mailbox.

The benchmark execution times exclude Android application startup and termination times. To reduce variability due to system and environmental factors, we minimize log output, disable

background processes, and utilize a temperature controlled testing environment (a refrigerator freezer) for all results except those presented in Section 4.2.6.

4.2.1 ***Pi Precision.*** The Pi precision benchmark [13] computes the value of *Pi* to a specified precision using a digit extraction algorithm. This benchmark uses a finite number of terms in the following formula to compute an approximation to the value of π:

$$\pi = \sum_{n=0}^{\infty} \left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6} \right) \left(\frac{1}{16} \right)^n$$

This benchmark uses a master-worker pattern with dynamic work distribution, in which the master sends more work (if available) to a worker that has completed its previous work (and notified the master accordingly by sending a reply with the result of the previous work).

Figure 4 shows the execution time of calculating Pi to 15,000 decimal places, for an increasing number of devices. Each device contains two worker actors, and the total amount of work remains constant as the number of devices is increased (strong scaling). These results were obtained using the Wi-Fi Soft AP based communication layer, and only a Nexus 5 device (not a Nexus 4 device) was used as the Soft AP for all configurations.

The chart in Figure 4 starts with a single Nexus 5 device running the Pi precision approximation, and each data point shows the execution time after adding one more device. After five Nexus 5 devices were added, we introduced three Nexus 4 devices incrementally for each of the remaining data points. We can observe proportional scaling with the first five Nexus 5 devices, with the scaling slowing down with the addition of Nexus 4 devices.

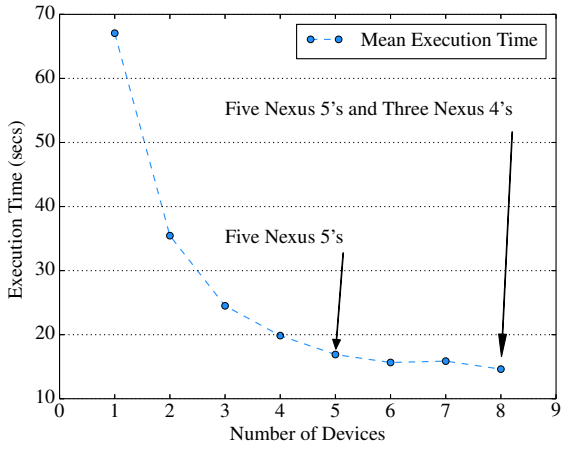

Figure 4: Pi Precision Computation: This figure shows the average execution time (over 20 executions) for computing the value of Pi to 15, 000 decimal places. The x-axis shows the number of devices used for the computation, and the y-axis shows the execution time. Each device runs two worker actors, and one device also runs a master actor. From one device to five devices the results are obtained using Nexus 5's, from six to eight devices, the additional devices are Nexus 4's.

This is because Nexus 4 devices are only half as powerful as Nexus 5 devices, adding modest increase in computing power, while still increasing the communication traffic to the AP host device. In spite of the limited computing capability of Nexus 4's, due to the dynamic nature of the generated work and the effective load balancing technique implemented by the application, the total execution time is still improved by adding slower Nexus 4's to the computation.

4.2.2 ***Cannon's Algorithm.*** Dense matrix multiplication is one of the most basic operations in scientific computations and is often at the core of many image processing algorithms. `Cannon's Algorithm` is a memory-efficient distributed matrix multiplication usually implemented on toroidal mesh interconnections [11]. It is designed for execution on a virtual $N \times N$ grid of processors, where matrices A and B are mapped onto the processors in a block-based fashion, with sub-blocks A_{ij} and B_{ij} mapped to processor p_{ij}.

The algorithm executes in two phases. In the first phase, the sub-blocks are aligned through an initial skew, where each sub-block A_{ij} is shifted left by some number of positions along the row and each sub-block B_{ij} is shifted up by some number of positions along the column. Each processor, p_{ij} receives blocks $A_{i,(j+i)modN}$ and $B_{(i+j)modN,j}$. The second phase is a series of circular shifts and computation of partial results. During each step, the sub-blocks are shifted one processor up or left, each processor multiplies the newly received sub-blocks and add the results to the sub-block C_{ij}, maintained at processor p_{ij}.

The loose coupling among processors and the message-passing nature of the algorithm makes it a natural candidate for an actor-based implementation. Our implementation of the Cannon's algorithm uses one selector to represent one independent processor. The scaling experiment is done on one to eight phones, with a matrix size of 1680×1680. As before, the first five devices are Nexus 5s, with three Nexus 4 devices added after that. To implement a $N \times N$ selector grid, each device hosts the same number of selectors as the number of devices in the network (e.g., for a three device network, each device hosts three actors). These results were obtained using the Wi-Fi Soft AP based communication layer, and only a Nexus 5 device (not a Nexus 4 device) was used as the Soft AP for all configurations.

Figure 5 shows the experimental results for a matrix size of 1680×1680 on one to eight mobile devices. For each experiment with N devices, each device hosts N actors to make an $N \times N$ processor grid in the original Cannon's algorithm. For one device, there will be a single processor, making the matrix multiplication serial in effect.

We can observe that the two device experiment achieves a 4× speedup, and the four device experiment achieves a 8× speedup compared to the serial execution. This is due to the fact that multiple actors on the same device also utilize intra-device multi-core computing power. As more devices are added to the network the performance improvement diminishes, due to the increased synchronization from the quadratically increasing number of processors. Further optimizations can be made with a more generalized matrix multiplication algorithm.

4.2.3 ***Trapezoidal Integration.*** `The Trapezoidal` benchmark [10, 13] approximates the integral function over an interval $[a, b]$

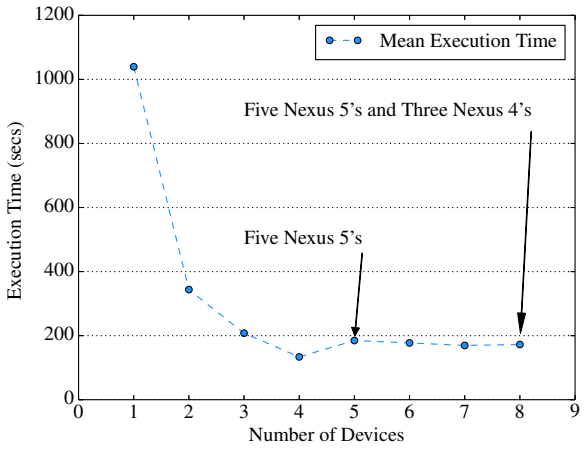

Figure 5: Cannon's algorithm: This figure shows the average time (over 20 executions) for multiplying two matrices of size = 1680×1680. For each experiment with N devices, each device hosts N actors (processors). The first five devices are Nexus 5s, and three Nexus 4 devices are added thereafter, as in Figure 4.

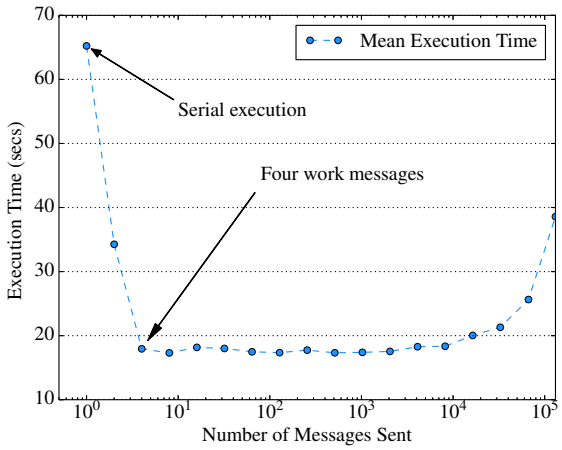

Figure 6: Trapezoidal Approximation: This figure shows the average time (over 20 executions) with 4 Nexus 5 devices to compute an approximation with 100, 000, 000 intervals for the integration. The x-axis (in log scale) shows the number of work messages sent by the master to workers, and the y-axis shows the average times. The total work remains the same for all experiments. For each work message, a reply message is sent back to the master with result.

by using the trapezoidal approximation [4, 31]. We approximate the integral of the function:

$$f(x) = \frac{1}{x+1} \times \sqrt{1 + e^{\sqrt{2x}}} \times \sin(x^3 - 1)$$

The parallelism is achieved by dividing up the integral approximation into a fixed number of intervals, by using a master-worker pattern. The original algorithm is obtained from [8], in which each worker is remotely created by the master actor, work is sent to each worker and the completed results are returned to the master.

We modified this implementation to use a request-reply model, where work is sent to workers piece by piece, and upon getting a result, the master will generate another piece of work and send to the replying worker when work is available. However, unlike the dynamic workload distribution in Section 4.2.1, the workload distribution in this benchmark remains fixed. The reason for a fixed distribution is because we use this benchmark to evaluate the impact of number of messages on performance, without considering the impact of dynamic load balancing. Since the workload is statically distributed, this benchmark is an excellent candidate for examining the tradeoff between the number of messages and workload size in each message. Note that each interval results in two messages exchanged between the master and a worker, a *work message* sent from the master to the worker and a *reply message* sent from the worker to the master.

In our experiments, we use a static network of 4 Nexus 5 devices, and increase the total number of work messages by powers of two, starting with 1 work message (serial execution). Figure 6 shows the number of work messages on the x-axis on a log scale, and shows the corresponding execution time in its y-axis.

We can observe that for a constant workload, the execution time becomes consistent once all four devices are involved (4 or more work messages are sent). For the results shown in Figure 6 (obtained for 10^8 intervals), the communication overhead does not affect the $4\times$ speedup until the point when 10^4 to 10^5 work messages are sent. This benchmark demonstrates the robustness of our system to effectively overlap communication latency with message processing, and shows that ideal parallelism can be achieved even with a relatively large number of communication messages.

4.2.4 ***Ping-Pong Microbenchmark.*** The `Ping Pong` microbenchmark involves two actors: a Ping actor sends a message to a Pong actor, which sends a message back in reply. The benchmark tests the message passing throughput among multiple devices in a mobile distributed environment. The original benchmark was obtained from [13] and implemented as a DAMMP application for this paper.

For the purpose of evaluating message-passing throughput and communication overhead, we conducted the experiment on four different configurations, each with two devices, where one device hosts a single ping actor, and another hosts a pong actor. (The four cases relate to whether the Ping and Pong actors are on Nexus 4 vs. Nexus 5 devices.) For this microbenchmark, we performed our experiment using both Wi-Fi AP and Wi-Fi Direct communication layers so as to compare their relative performance.

Table 1 shows the round trip delay time for a single pair of ping-pong messages. To obtain a better estimate of the underlying message-passing latencies, we measured the round-trip delays by instrumenting the Communication Manager supporting the Ping Actor, so as to exclude delays due to processing other messages at the Ping Actor level. From these numbers, we see that the round trip latencies are consistently smaller for Wi-Fi AP relative to Wi-Fi Direct based communications.

Finally, we performed throughput measurements by using a sliding window of 5,000 messages. The throughput measurements are taken with multiple configurations with our test devices with Wi-Fi based based communication layers. The results are shown in Table 2.

No. of Devices		Avg. Msg Round-trip Time (ms)	
Nexus 5	Nexus 4	Wi-Fi AP	Wi-Fi Direct
2	0	583	7,144
0	2	403	1,696
1(H)	1	2,746	4,249
1	1(H)	105	892

Table 1: Average round trip latency for a single pair of ping-pong messages. The measurement is taken as the time between a ping message sent by the Communication Manager for the Ping actor and its corresponding pong message (marked by the same message ID) is received back in the Communication Manager. The average was performed over 250 round trip message pairs. The (H) suffix identifies which device acts as the Soft AP (host device) and implements the Ping actor.

No. of Devices		Wi-Fi AP Throughput (msg/sec)		
Nexus 5	Nexus 4	Arith. Mean	Best	Worst
2	0	4898.745	5202.433	4640.597
0	2	3239.891	3382.106	3037.053
1(H)	1	3279.514	3500.460	3118.071
1	1(H)	3680.513	3827.222	3550.441

Table 2: Wi-Fi AP throughput: this benchmark tests the message throughput in distributed Android applications with Wi-Fi AP based communication layer. The ping actor sends a total of 50, 000 **messages with a sliding window of** 5, 000 **messages and terminates after it receives the** 50, 000 **corresponding pong messages. Throughput is calculated by dividing the total number of messages sent (**100, 000**) by the benchmark execution time.**

In the Wi-Fi based environment (see Table 2, rows 1 and 2), where the ping actor acts as a Soft AP host and devices connect to each other through direct TCP/IP links, the message throughput remains consistent over networks with multiple homogenous devices. The Nexus 5 message throughput is expectedly 2× better than Nexus 4 given its more powerful SoC. However, in a heterogeneous setup (see Table 2, rows 3 and 4) where one model acts as the Soft AP and hosts a ping actor and the other device hosts a pong actor, we observe some discrepancies. Even with less powerful hardware, the Nexus 4 device exhibits a better throughput when acting as a Soft AP host.

4.2.5 ***Computation offloading.*** To illustrate the usability of our adaptive offloading model and how it can benefit the application performance, we provide an experimental evaluation using a two player strategy board game, Othello. We evaluated this game on a Nexus 5 mobile device with a MacBook Pro laptop available for computation offloading.

Othello. A two player (Player 1 and Player 2) strategy board game, played on a 8 × 8 uncheckered board. Each player has a designated color or symbol [1] for discs placed on the board. For example, lets say that Player 1 is assigned the symbol 'X' and Player 2 is assigned the symbol 'O'. Each player takes a turn by placing an 'X' or 'O' on the board based on it's assigned symbol. During the game, any discs of the opponent player's color which are in a straight line, and bounded by the disc just placed on the board and another colored disc of the current player, are switched to the current player's color. The objective of the game is for a majority of the discs to display your color when the last playable empty square is filled [34]. For our evaluation, we mimic one player as a human and the other player as an AI algorithm. The AI can look ahead upto six steps to decide it's next move. Computing all the board combinations with a look ahead of six can be very-time consuming on a mobile device.

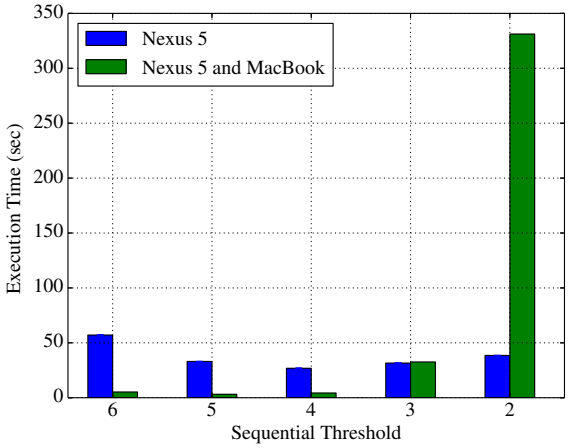

Figure 7: Execution time of the AI to find the best move with a lookahead depth of six. Evaluations have been performed on without offloading on a Nexus 5, and with offloading to a MacBook, while the game was being played on the Nexus 5. Lower is better.

Figure 7 shows the execution time for the Othello game, where the y-axis shows the execution time in seconds and the x-axis is the *sequential threshold* for the lookahead computation. Since each move is processed in parallel in a recursive call, there can be a lot of overhead incurred in the creation of and execution of leaf-level fine-grained tasks. To optimize this algorithm, we introduce a sequential threshold, and upon reaching that threshold all moves will be processed sequentially. We start with threshold set to six, hence all six recursive calls for the look ahead will be performed sequentially. The blue bar denotes the execution time of the AI, when both players are running on the Nexus 5 and the green bar denotes the execution time (calculated on the phone) of the AI computation, after the completion of the offloaded computation to the MacBook with the same lookahead. In this evaluation, the offloading has been done based on the sequential threshold. We observe a significant speedup due to offloading of around 7× - 10× with threshold values ranging from six to four. Smaller threshold values of two and three do not show an improvement due to offloading because the offloading overheads dominate any potential benefits in those cases.

4.2.6 ***Thermal effect of adaptive offloading.*** We performed two experiments with: *a*) the *trapezoidal benchmark* (Section 4.2.3), and, *b*) the *Pi Precision benchmark* (Section 4.2.1), without temperature control (i.e., without placing the devices in a freezer) in a more

[1] We will use the terms, "color" and "symbol" interchangeably in this description

realistic usage scenario for mobile hand held devices, to show the impact of thermal dissipation on the application performance and the device.

Trapezoidal Benchmark. For the *trapezoidal* benchmark, we used the same implementation as in Section 4.2.3 to calculate the area of the integral function with 100, 000, 000 intervals for the integration.

Figure 8 shows the effect of running the benchmark on one Nexus 5. The y-axis shows the throughput (calculated for every 32768 points per sec) and the mean temperature (calculated using 11 on-device sensors [28]) of the device. The shared x-axis shows the execution time for the completion of the benchmark.

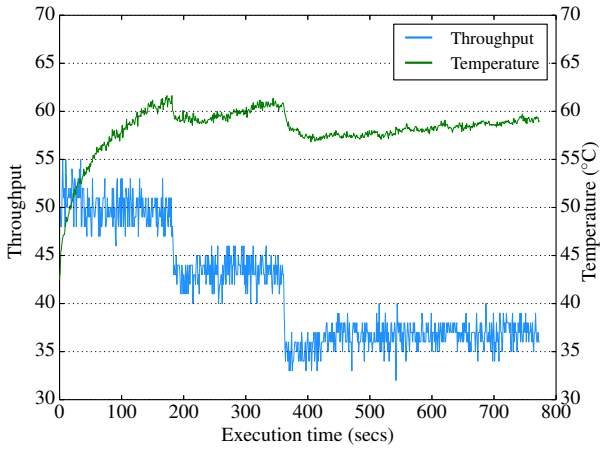

Figure 8: Trapezoidal approximation: Computing the area of an integral function with 100, 000, 000 **intervals for the integration on a single Nexus 5 device.**

One must note, that the throughput of the application constantly drops when the device reaches approximately $60°C$ as the operating system on Nexus 5 downclocks the processor to prevent the device from overheating. Since the compute intensive application continues running on the phone, the temperature holds at $60°C$ but the throughput keeps dropping.

Figure 9 shows the effect or running the same trapezoidal approximation benchmark with the same configuration, but using one Nexus 5 and the MacBook Pro for offloading. We used our adaptive offloading model to offload the entire computation *automatically* to a powerful device (MacBook Pro in our case), when the temperature reaches $60°C$. We used the *temperature message* to obtain the temperature of the current device and all other devices connected to our ad-hoc network, and offloaded the computation (using Wi-Fi) when it reached the user-defined threshold.

One must note, that in this case, the throughput increases after we offload the computation at $55°C$. All measurements are performed on the Nexus 5 device. It is also important to note that as the computation proceeds on the offloaded device, the Nexus 5 device reaches the normal temperature threshold ($40°C - 50°C$). We observe almost a 5× speedup on offloading the computation to

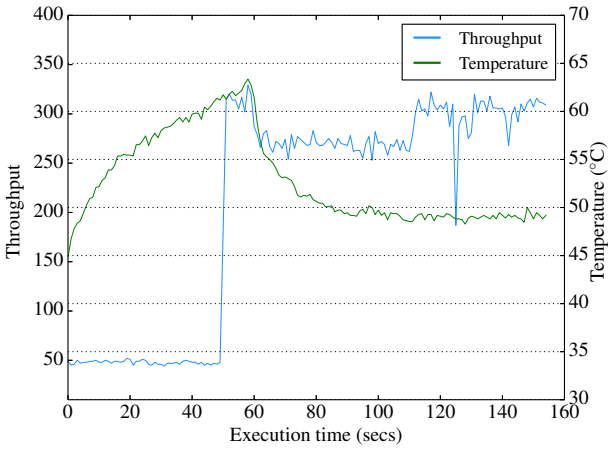

Figure 9: Trapezoidal approximation: Computing the area of the integral function with 100, 000, 000 **intervals for the integration on a Nexus 5 device. Computation is automatically offloaded to a MacBook Pro when the temperature on the Nexus 5 reaches** $60°C$**, using our adaptive offload model.**

the MacBook Pro (even though the execution time of the application includes the network overhead for offloading the computation).

Pi Precision Benchmark. The Pi precision benchmark computes an approximation of the value of Pi to a specified precision using a digit extraction algorithm as explained in Section 4.2.1.

Table 3 shows the thermal difference on the master device for five iterations of the Pi precision benchmark, with the value of Pi calculated to 15, 000 decimal points. The total execution time for five iterations are recorded and the temperature difference shown in the table. We use a total of five Nexus 5 and three Nexus 4 device which are fully charged and at room temperature at the beginning of each experiment. The temperature data are taken at the begin and end of each experiment on the device that hosts the Soft AP and master actor, on the 11 on-chip thermal sensors in the devices. The arithmetic mean is calculated by the difference between average of 11 sensors at beginning and end of the experiment, the maximum and minimum temperature difference are calculated with individual sensor data [28].

We can observe the temperature increase goes down for the host device, as more devices are added to the network for work offloading, showing that offloading across multiple devices reduces the thermal impact of compute-intensive applications.

As discussed in [35], the thermal behavior of the mobile SoC demonstrates complex behavior affected by both the application processor and the battery. The power consumption on mobile devices, can also be heavily impacted by thermal dissipation [25]. As multiple scheduling and power management techniques are developed with the thermal limit in consideration, past work has concentrated on thermal dissipation control through software based task scheduling [7], and architecture based improvements [16]. Our experiments show a new possibility for thermal-aware applications by offloading computationally intensive tasks to nearby devices with a small communication cost.

Nexus 5	Nexus 4	Total Exec. Time on Host (sec)	Temperature on the host device (°C)		
			Arith. Mean	Max	Min
1	0	448.259	16.82	–	–
2	0	253.818	13.205	22.18	12.91
3	0	163.376	15.850	22.27	13.91
4	0	124.659	16.113	19.73	13.73
5	0	104.100	16.144	20.64	14.73
5	1	91.793	14.940	19.64	12.00
5	2	88.566	13.817	19.45	10.27
5	3	75.202	12.906	15.00	10.45

Table 3: Pi Precision benchmark under a realistic usage scenario. Each experiment is executed with 5 iterations and total execution time is recorded on the Soft AP host device (Nexus 5).

5 RELATED WORK

Ever since its inception, the logical isolation, and asynchrony inherent in the actor model has made it an attractive candidate for distributed computing for decades, with more recent explorations of the actor model for distributed mobile platforms. We briefly summarize some related work on using the actor model for distributed mobile computing in recent years.

AmbientTalk. The language is an actor-based programming language designed specifically for `mobile ad hoc networks` [29, 30] It features λ calculus based functional elements with local and remote actor-based reductions, and object-oriented elements with both parameter pass-by-value (isolated objects) and pass-by-reference "regular" objects) semantics. It utilizes the Actor model for its concurrency and distributed computation. Actors in the `AmbientTalk` VM are used as containers to hold a set of regular objects, rather than regular "active objects". The virtual machine hosts multiple actors that can execute concurrently, while each actor itself represents a communicating event loop that uses the *run-to-completion* semantics for method invocation on its host objects. While the `AmbientTalk` model has similarities with the traditional actor model, one difference is that it can break the pure message-passing model through the use of *far references* introduced in the E language [22].

The `AmbientTalk` language has both cluster-based and early Android implementations that focus on high-level abstractions for distributed programming with both message passing mechanisms and remote accesses through far references. In our work, on the other hand, the focus is on supporting a pure actor/selector model at the high-level, with distributed mechanisms supported in configurations that are decoupled from the programming application logic. Further, our model does not limit ourselves to *mobile ad-hoc networks* since it can also support communications within and across mobile devices and server devices with a single model. Finally, to the best of our knowledge, `AmbientTalk`'s implementation targets an outdated version of Android (prior to Android 4.0) that is no longer supported on current mobile devices, thereby preventing us from performing experimental comparisons with `Ambient Talk`.

ActorDroid. This project is based on SCALA actors and focuses on a distributed application framework which follows the stream processing paradigm inherited from SCALA [23]. The framework's basic units are *services*, each independently a SCALA actor that runs in its execution environment. It uses a publish-subscribe model for service discovery and join, while each mobile device can host one-to-many services.

The `ActorDroid` work described an implementation based on external Wi-Fi Access Points with dynamic network maintained by a Master-Worker scheme. We also use a decentralized structure in our runtime to enable support for dynamic topologies. However, unlike `ActorDroid`, the hardware dependent communication layer is completely decoupled from the application-level actor communication in our approach. Instead of relying on a pre-defined strategy for dynamic topological changes, we expose a minimal amount of information to application-level actors that can provide the user-level application with the ability to adapt to runtime events. As with `Ambient Talk`, to the best of our knowledge, `ActorDroid`'s implementation targets an outdated version of Android (prior to Android 4.0) that is no longer supported on current mobile devices, thereby preventing us from performing experimental comparisons with `ActorDroid`.

ActorNet. This project is an actor-based mobile agent platform for wireless sensor networks (WSNs) [18]. This project aims to create a high-level abstraction for concurrent and asynchronous programming for WSNs to adapt to the limited hardware resources available on mobile sensors. The ActorNet project implements a `Scheme` interpreter that is assumed to be better suited to the limited processing power and memory available on wireless sensors than stack based virtual machines, and introduces new language primitives that enable actor-based message passing, queries, and access to program continuation.

The `ActorNet` project also focuses on higher level language abstractions to aid parallel and asynchronous programming on mobile networks. Compared to the emphasis on optimization for limited hardware specifically for wireless sensors, our work focuses more on adaptability on a wider range of mobile devices by creating a more general distributed runtime system. Our design goes beyond sensor networks and supports rich combinations of mobile devices and cluster-based services. Compared to `ActorNet`, our design is better suited for use with more powerful consumer mobile devices such as high-end tablets and smartphones. Finally, since `ActorNet` provides an implementation specific to wireless sensors, we were unable to perform experimental comparison with `ActorNet`.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we designed and implemented multiple extensions to the Java-based HJDS distributed-selector runtime for servers, so as to obtain the DAMMP library for distributed Android devices. Our work focuses on decentralized distributed applications using the actor / selector model by supporting a highly decoupled and customizable communication middleware, and support for application-level runtime event handling. We provide a hierarchical, heterogeneous concurrency and distribution model by extending the actor model from HJDS to support mobile devices. We also

introduced a task offloading pattern based on the selector model and the Master-Worker paradigm.

We demonstrated the scalability of computationally intensive applications on distributed mobile platforms, examined the message passing overheads with two promising off-the-grid wireless communication technologies, and showed decreased thermal dissipation from offloading, while maintaining scalability for compute-intensive applicatons in a realistic ad-hoc mobile network environment.

For future work, with the empirical results in mind, we plan to explore real-world heuristics in thermal-aware dynamic distribution on heterogeneous mobile networks that involve devices with various computing powers, including wireless sensors, tablets, and laptops. We also plan to explore dynamic load-balancing across devices by having the runtime automatically migrate Actors when needed for various reasons (maximizing overall performance, maximizing combined system battery life, etc.).

REFERENCES

[1] Gul Agha. 1986. *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA.

[2] Android Developers. 2014. ART and DALVIK. (2014). https://source.android.com/devices/tech/dalvik/

[3] Android Developers. 2017-01-24. Wi-Fi Peer-to-Peer. (2017-01-24). https://developer.android.com/guide/topics/connectivity/wifip2p.html

[4] John Ayres and Susan Eisenbach. 2009. Stage: Python with Actors. In *Proceedings of IWMSE '09*. IEEE Computer Society, Washington, DC, USA, 25–32.

[5] Arani Bhattacharya and Pradipta De. 2017. A survey of adaptation techniques in computation offloading. *Journal of Network and Computer Applications* 78 (2017), 97–115.

[6] D. Camps-Mur, A. Garcia-Saavedra, and P. Serrano. 2013. Device-to-device communications with Wi-Fi Direct: overview and experimentation. *IEEE Wireless Communications* 20, 3 (June 2013), 96–104. DOI: https://doi.org/10.1109/MWC.2013.6549288

[7] Thidapat Chantem, Robert P. Dick, and X. Sharon Hu. 2008. Temperature-aware Scheduling and Assignment for Hard Real-time Applications on MPSoCs. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '08)*. ACM, New York, NY, USA, 288–293. DOI: https://doi.org/10.1145/1403375.1403446

[8] Arghya Chatterjee, Branko Gvoka, Bing Xue, Zoran Budimlic, Shams Imam, and Vivek Sarkar. 2016. A Distributed Selectors Runtime System for Java Applications. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '16)*. ACM, New York, NY, USA, Article 3, 11 pages. DOI: https://doi.org/10.1145/2972206.2972215

[9] Charalampos Doukas and Ilias Maglogiannis. 2010. A fast mobile face recognition system for android OS based on Eigenfaces decomposition. In *IFIP International Conference on Artificial Intelligence Applications and Innovations*. Springer, 295–302.

[10] EPCC. 2001. The Java Grande Forum Multi-threaded Benchmarks. (2001). http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/threads/s1contents.html

[11] H. Gupta and P. Sadayappan. 1994. *Communication Efficient Matrix-Multiplication on Hypercubes.* Technical Report 1994-25. Stanford Infolab. http://ilpubs.stanford.edu:8090/59/

[12] Shams Imam and Vivek Sarkar. 2014. Habanero-Java Library: A Java 8 Framework for Multicore Programming. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '14)*. ACM, New York, NY, USA, 75–86. DOI: https://doi.org/10.1145/2647508.2647514

[13] Shams Imam and Vivek Sarkar. 2014. Savina - An Actor Benchmark Suite. In *Proceedings of the 4th International Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE! 2014)*.

[14] Shams M. Imam and Vivek Sarkar. 2014. Selectors: Actors with Multiple Guarded Mailboxes. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents and Decentralized Control (AGERE! '14)*. ACM, New York, NY, USA, 1–14. DOI: https://doi.org/10.1145/2687357.2687360

[15] K. Jahed, O. Farhat, G. Al-Jurdi, and S. Sharafeddine. 2016. Optimized group owner selection in WiFi direct networks. In *2016 24th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. 1–5. DOI: https://doi.org/10.1109/SOFTCOM.2016.7772169

[16] O. Khan and S. Kundu. 2009. Hardware/software co-design architecture for thermal management of chip multiprocessors. In *2009 Design, Automation Test in Europe Conference Exhibition*. 952–957. DOI: https://doi.org/10.1109/DATE.2009.5090802

[17] Mikkel Baun Kjærgaard, Jakob Langdal, Torben Godsk, and Thomas Toftkjær. 2009. EnTracked: Energy-efficient Robust Position Tracking for Mobile Devices. In *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services (MobiSys '09)*. ACM, New York, NY, USA, 221–234. DOI: https://doi.org/10.1145/1555816.1555839

[18] YoungMin Kwon, Sameer Sundresh, Kirill Mechitov, and Gul Agha. 2006. ActorNet: An Actor Platform for Wireless Sensor Networks. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '06)*. ACM, New York, NY, USA, 1297–1300. DOI: https://doi.org/10.1145/1160633.1160871

[19] J. S. Lee, Y. W. Su, and C. C. Shen. 2007. A Comparative Study of Wireless Protocols: Bluetooth, UWB, ZigBee, and Wi-Fi. In *IECON 2007 - 33rd Annual Conference of the IEEE Industrial Electronics Society*. 46–51. DOI: https://doi.org/10.1109/IECON.2007.4460126

[20] C. Lindemann and O. P. Waldhorst. 2002. A distributed search service for peer-to-peer file sharing in mobile applications. In *Proceedings. Second International Conference on Peer-to-Peer Computing,*. 73–80. DOI: https://doi.org/10.1109/PTP.2002.1046315

[21] Richard K. Lomotey, Yiding Chai, Ashik K. Ahmed, and Ralph Deters. 2013. Distributed Mobile Application for Crop Farmers. In *Proceedings of the Fifth International Conference on Management of Emergent Digital EcoSystems (MEDES '13)*. ACM, New York, NY, USA, 135–139. DOI: https://doi.org/10.1145/2536146.2536174

[22] Mark S. Miller, E. Dean Tribble, and Jonathan Shapiro. 2005. *Concurrency Among Strangers.* Springer Berlin Heidelberg, Berlin, Heidelberg, 195–229. DOI: https://doi.org/10.1007/11580850_12

[23] Pierre-André Mudry and Romain Cherix. 2012. ActorDroid - A distributed computing framework for mobile devices based on SCALA actors. *ScalaDays* (2012).

[24] Abdul Mutholib, Teddy Surya Gunawan, and Mira Kartiwi. 2012. Design and implementation of automatic number plate recognition on android platform. In *Computer and Communication Engineering (ICCCE), 2012 International Conference on*. IEEE, 540–543.

[25] Umit Y. Ogras, Raid Z. Ayoub, Michael Kishinevsky, and David Kadjo. 2013. Managing Mobile Platform Power. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD '13)*. IEEE Press, Piscataway, NJ, USA, 161–162. http://dl.acm.org/citation.cfm?id=2561828.2561861

[26] C Muthu Ramya, M Shanmugaraj, and R Prabakaran. 2011. Study on ZigBee technology. In *Electronics Computer Technology (ICECT), 2011 3rd International Conference on*, Vol. 6. IEEE, 297–301.

[27] Carlo Ratti, Dennis Frenchman, Riccardo Maria Pulselli, and Sarah Williams. 2006. Mobile landscapes: using location data from cell phones for urban analysis. *Environment and Planning B: Planning and Design* 33, 5 (2006), 727–748.

[28] Sujith Thomas, Zhang Rui. 2014. Generic Thermal Sysfs driver How To. (2014). https://www.kernel.org/doc/Documentation/thermal/sysfs-api.txt

[29] Tom Van, Cutsem Christophe, Scholliers Dries Harnie, and Wolfgang De Meuter. *An Operational Semantics of Event Loop Concurrency in AmbientTalk.* Technical Report.

[30] Tom Van Cutsem, Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Dries Harnie, Kevin Pinte, and Wolfgang De Meuter. 2014. AmbientTalk: programming responsive mobile peer-to-peer applications with actors. *Computer Languages, Systems & Structures* 40, 3 (2014), 112–136.

[31] Carlos Varela and Gul Agha. 2001. Programming Dynamically Reconfigurable Open Systems with SALSA. *ACM SIGPLAN Notices* 36, 12 (2001), 20–34.

[32] Ming-Heng Wang Ph D and others. 2012. Feasibility of using cellular telephone data to determine the truckshed of intermodal facilities. (2012).

[33] Sarah E Wiehe, Aaron E Carroll, Gilbert C Liu, Kelly L Haberkorn, Shawn C Hoch, Jeffery S Wilson, and JDennis Fortenberry. 2008. Using GPS-enabled cell phones to track the travel patterns of adolescents. *International journal of health geographics* 7, 1 (2008), 22.

[34] Wikipedia. June 2017. Othello/Reversi. (June 2017). https://en.wikipedia.org/wiki/Reversi

[35] Qing Xie, Jaemin Kim, Yanzhi Wang, Donghwa Shin, Naehyuck Chang, and Massoud Pedram. 2013. Dynamic Thermal Management in Mobile Devices Considering the Thermal Coupling Between Battery and Application Processor. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD '13)*. IEEE Press, Piscataway, NJ, USA, 242–247. http://dl.acm.org/citation.cfm?id=2561828.2561877

[36] Li Zheng. 2006. ZigBee wireless sensor network in industrial applications. In *SICE-ICASE, 2006. International joint conference*. IEEE, 1067–1070.