Swing: Swarm Computing for Mobile Sensing

Songchun Fan*
Google
Mountain View, CA, USA
Email: schfan@google.com

Theodoros Salonidis
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
Email: tsaloni@us.ibm.com

Benjamin Lee
Duke University
Durham, NC, USA
Email: benjamin.c.lee@duke.edu

***Abstract*—This paper presents Swing, a framework that aggregates a swarm of mobile devices to perform collaborative computation on sensed data streams. It endows performance and efficiency to the new generation of mobile sensing applications, in which the computation is overly intensive for a single device. After studying the source of performance slowdown of the sensing applications on a single device, we design and implement Swing to manage (i) parallelism in stream processing, (ii) dynamism from mobile users, and (iii) heterogeneity from the swarm devices. We build an Android-based prototype and deploy sensing apps – face recognition and language translation – on a wireless testbed. Our evaluations show that with proper management policies, such a distributed processing framework can achieve up to 2.7x improvement in throughput and 6.7x reduction in latency, allowing intensive sensing apps to reach real-time performance goals under different device usages, network conditions and user mobility.**

I. Introduction

Emerging mobile applications involve continuous sensing and complex computations on sensed data streams. Examples include cognitive applications (e.g., speech recognition, natural language translation, as well as face, object, or gesture detection and recognition) and anticipatory applications that proactively track and provide services when needed.

Unfortunately, today's mobile devices cannot keep pace with such applications, despite advances in hardware capability. Consider as an example a mobile face recognition application. Our experiments show that an individual mobile device is insufficient for this app, as it requires continuous, intense computation on sensed image streams. Figure 1 illustrates this observation as various mobile devices process video frames for face recognition. Each device can only process 4~10 frames per second (FPS), which is far below the minimal 24 FPS required for smooth video playback. Over time, mismatched arrival and processing rates cause new frames to queue and end-to-end delays to increase. Even the fastest device (*H*, a quad-core LG Nexus 5) fails to keep up — its end-to-end frame delay increases to 1.2s after only 5s of computation! Even if a user could perform the computation by herself, the energy burden would be unbearable. We observe that the camera-based face recognition app exhausts a fully charged phone battery in about two hours, with 40% of the energy consumed by computation.

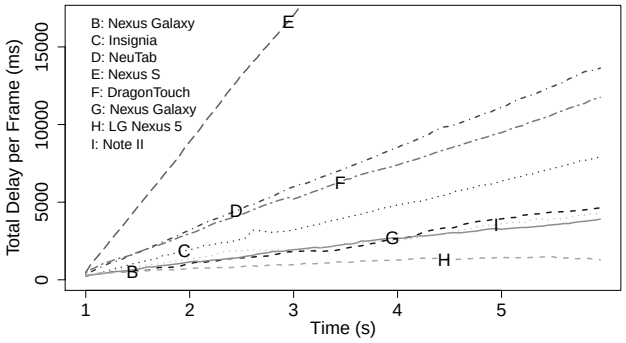

Fig. 1: Delay per frame when processed on different phones at 24 FPS load. Delays build up rapidly, and different phones have different reactions to the same load.

Traditional approaches address this problem by computation offloading. One approach offloads by sending sensed streams to remote cloud servers via cellular networks or to cloudlets via Wi-Fi, where a clone of the app runs [10], [11], [26]. However, cloudlets may not be widely deployed and access to cloud infrastructure may yield high network delays and can be intermittent due to mobility. A second approach offloads to local accelerators by rewriting code to use the DSP or GPU within mobile devices. However, using accelerators requires nontrivial programming effort and produces widely varying results for diverse codes on heterogeneous devices. In addition, it is not based on a high level programming model and is not portable to a wide range of IoT devices.

We adopt a different approach that looks beyond a single device but beneath the cloud. Today's users often carry multiple mobile devices (e.g., smart watches, phones and tablets) that provide redundant computing resources. This trend will increase with the explosive IoT growth of wireless devices in many shapes and forms. In such situations, idle devices could collaborate and receive offloaded computation. Furthermore, additional offload opportunities exist when multiple co-located users share a common sensing and computation goal. For example, a security team that patrols a route can collaboratively sense and analyze the video for face recognition, or a group of travelers could benefit from real-time translation of native speakers using collaborative processing on their mobile devices.

*This work was conducted while the author was an intern at the IBM T.J. Watson Research Center and a PhD student at Duke University.

In such usage scenarios and many others, Internet connectivity can be sparse and users may prefer to rely on local assets rather than the cloud [10], [11]. Moreover, the mobility of the users may forbid the usage of local stationary infrastructures such as cloudlet [27], [26]. Such circumstances leave users with no choice but either run the app entirely on their own devices, or collaboratively using the surrounding devices. If several proximate users demand results from the same application, collaborative computing can reduce the energy costs for any one user. Moreover, it mitigates the limitations of individual devices and improves service quality for all users.

We present Swing (SWarm computing for mobile sensING), a framework that aggregates mobile devices to collaboratively compute a shared answer. Swing views a collection of nearby mobile devices as a *swarm*. Swing does not require cloud assistance for executing mobile apps. It uses a dataflow programming model and Java-based API, which enables developers to rapidly build Swing applications. A central issue in such an architecture is meeting real-time requirements while coping with device heterogeneity and dynamics of the mobile environment. Swing provides a distributed resource management framework that aims to minimize processing latency while reducing resource usage and energy consumption. It also provides mechanisms for automatic device discovery, dynamic application deployment and mobility handling.

Specifically, our contributions are as follows:

1) **System architecture.** We design and implement a novel distributed computing framework for compute-intensive mobile sensing applications. We detail Swing's programming model and workflows which implement data stream processing for mobile swarms. Swing APIs allow sophisticated sensing apps with complex computations to be developed in a rapid and flexible manner. In contrast to existing approaches, Swing is a collaborative mobile data stream processing platform that does not require Internet connectivity. In addition, since it is Java-based it can be ported to a wide variety of IoT devices.

2) **Distributed resource management policies.** Swing enables a wide spectrum of resource management policies for various performance objectives and handles device joining and leaving, as well as user mobility. We propose a distributed algorithm that aims for minimization of latency and energy usage subject to a performance target. The algorithm distributes stream data dynamically across mobile devices, and manages data flow according to heterogeneous device capability.

3) **Detailed evaluation with real sensing apps and wireless testbed.** On our Android-based Swing prototype, we deploy two sensing apps – face recognition and language translation – on a wireless testbed with up to nine heterogeneous, mobile phones and tablets. We demonstrate the performance heterogeneity of these devices and evaluate the performance of the framework under various scenarios such as user moving, leaving and joining the system at runtime. Collectively, our results show that Swing efficiently and reliably manages multiple devices to meet the real-time performance goals of modern sensing apps with negligible overhead. Compared with a state of the art algorithms in existing stream processing systems, it achieves a 2.7x improvement in throughput and 6.7x reduction in latency.

The paper is structured as follows. In Section II, we provide a summary of related work. In Section III, we demonstrate the challenges in designing a distributed mobile computing framework. In Section IV and V, we introduce the system design and implementation. Finally in Section VI, we evaluate the performance of system with respect to different usage scenarios.

II. Related Work

Mobile computation offloading and edge computing. Existing mobile cloud computing frameworks such as CloneCloud and MAUI partition and offload mobile code to the cloud [10], [11]. Cloud offloading techniques cannot easily support real-time applications due to high delays between mobile and cloud.

Cloudlets [27] and edge computing reduce the delay by bringing server infrastructure closer to the mobile devices (e.g. LAN or WLAN level) and have been shown to support real time applications [26], [31]. Recent works utilize edge computing to bypass offloading to the cloud. In [14] edge gateways act as compute caches and process mobile face recognition queries directed toward the cloud. The work in [29] proposes a solution for distributing deep learning inference computations vertically between device, edge and cloud infrastructure. These approaches address vertical distribution of computations assuming a cloud/edge infrastructure. Swing focuses on horizontal parallel processing among multiple devices and does not assume such a hierarchical infrastructure.

One issue with cloudlet and edge computing approaches is that they require relatively costly investment in bringing compute infrastructure close to all mobile devices. They may exist in locations where a specific real-time application is needed but may not be as widely available as proximal mobile phones, especially in collaborative ad hoc applications targeted by Swing. Nevertheless, Swing does support "cloudlet mode" through Android virtual machines if a cloudlet infrastructure is available.

Industry solutions. A recent industry trend is for cloud providers to extend their IoT services to the network edge [7], [16]. These solutions use an edge gateway to support data aggregation from IoT devices or control-plane operations such as deployment and configuration of data analytics to IoT devices. In contrast, Swing enables more complex sensing applications such as video analytics that exploit parallel computation on multiple heterogeneous mobile devices which can be more powerful than IoT sensors. Tensorflow [5] is a recent platform which expresses deep learning models as dataflow graphs. It is primarily designed for GPUs and data center environments but also

Phone ID	B	C	D	E	F	G	H	I
Model	GalaxyNexus	Insignia7	NeuTab7	GalaxyS	DragonTouch	GalaxyNexus	Nexus4	Note2
Processing Delay (ms)	92.9	121.6	167.7	463.4	166.4	82.2	71.3	78.0
Throughput (FPS)	10	8	6	2	5	12	13	12

TABLE I: Performance Heterogeneity

supports deployment on mobile devices. Swing is designed for mobile devices and focuses on general-purpose computation and sensing applications than deep learning. In order to support efficient training of deep models, Tensorflow supports offline mechanisms for optimizing placement of a deep model dataflow graph stages to processors. However, it does not provide real-time resource management mechanisms that handle device heterogeneity and mobility.

Data stream processing systems. Swing uses a dataflow graph computation model similar to data stream processing systems [1], [9], [36], [2], [3]. Such systems typically process a large number of data streams inside compute clusters or data centers. Sonora [34] is a stream processing system that can support processing on mobile phones. However, it is based on a client-server model and does not support adaptive offloading of stream computations between mobile and cloud. In contrast, we use the dataflow computation model to program, decompose and distribute collaborative mobile sensing applications on multiple mobile devices. Unlike Sonora which assumes that a computation task can enjoy infinite compute capacity once it is offloaded to the server, our framework utilizes surrounding mobile devices which incur challenges in resource management.

Delay-oriented resource management in distributed systems. Resource management aiming at delay optimization has been extensively addressed in a variety of distributed systems. Scheduling algorithms based on bin packing for single-stage parallel systems [6] or based on HEFT heuristic for real-time graph-based applications [35]. Other approaches solve the problem of mapping a graph of execution tasks to an underlying network of machines [20], [12]. Most approaches above require a-priori knowledge of task execution times and global knowledge of network state. Distributed scheduling heuristics that estimate task execution times and avoid stragglers have been developed for big data platforms (e.g. Mapreduce and Spark) [24]. These approaches are for batch systems that schedule a finite set of tasks on thousands of machines in a data center as opposed to infinite data streams on a finite set of mobile devices. None of the above approaches aims for minimization of system energy consumption. Existing stream processing systems specialize on data streams but typically use round robin scheduling because it is adequate for data center environments [9], [2], [3]. Recent mobile stream processing systems [23] do not provide resource management mechanisms that address mobile heterogeneity and dynamics. As will show in the evaluation round robin is not adequate for mobile environments. In summary, the resource management of Swing differs from previous work on distributed systems in that it operates on data streams, uses a distributed algorithm, and aims for delay minimization with a minimum number of compute resources (which aims to minimize energy consumption).

Distributed and shared mobile computing. Pocket Switched Networks [19], Throwboxes [8], and ferry-based networks [17] are distributed mobile frameworks that focus on efficient communication in mobile networks with intermittent connectivity as opposed to efficient collaborative mobile computation on sensed data streams. Serendipity [28] is a theoretical framework for delay-optimized task allocation in mobile networks with intermittent connectivity. The algorithm greedily selects a single processor over a network path of minimum estimated task completion time and is evaluated using simulations. Swing focuses on real-time parallel processing among multiple connected devices and is evaluated using a real system implementation. Medusa [25], a crowd-sensing framework for mobile users, is targeted toward data collection of crowd-sensed data from multiple devices to a cloud server, rather than real-time collaborative computations among mobile devices. Misco [13], Hyrax [21], CWC [6] implement MapReduce-like frameworks for parallel task execution on mobile phones. MapReduce caters to a batch processing model rather than a real-time computation model targeted by our work. MobiStreams [32] provides a distributed stream processing runtime for mobile phones and focuses on the orthogonal issue of fault tolerance. It does not provide any mechanisms for efficient sharing of execution load among the devices and requires the assistance of a cellular network and a centralized server in the cloud for coordination. In contrast, our framework relies purely on mobile devices and can utilize mobile hotspot APs, Wi-Fi Direct, WLAN or cellular, as networking technologies.

III. Challenges

We identify two major factors that challenge the design of a distributed mobile data stream processing platform: device heterogeneity and dynamism. In this section, we quantify their impact with preliminary experiments.

Today's mobile devices deploy a broad spectrum of hardware. First, we characterize this inherent performance heterogeneity with the following experiments. We use nine devices – *A*: Galaxy S3, *B*: Galaxy Nexus, *C*: Insignia7 tablet, *D*: NeuTab7 tablet, *E*: Galaxy S, *F*: DragonTouch tablet, *G*: Galaxy Nexus, *H*: LG Nexus4, *I*: Galaxy Note2. All devices are connected to a wireless router (Linksys E1200 802.11n 2.4GHz channel 1) and located in the same office. In each

experiment, phone A sends 24 FPS video frames over a strong Wi-Fi connection to another phone $i \in \{B, C, \cdots, I\}$, which conducts face recognition. Each i measures and records the processing delay of each frame. Experiments are conducted during the night to reduce chances of interference from other wireless communications. Each experiment runs for 10 minutes (14400 video frames).

Device heterogeneity. The second line in Table I reports the average processing delay per frame (excluding queuing delay) for each phone. The third line in Table I reports the corresponding throughputs (inverse delays) which represent the computational capacity of each device. We observe high performance heterogeneity: the fastest phone H reports throughput that is 6 times higher than that of the slowest phone E. However, even H cannot provide a throughput that is as high as the input rate of 24 FPS.

Achieving high throughput using multiple devices is also challenging due to the existence of *stragglers*, i.e. devices that either are slow or are on slow wireless network links. Without proper resource management, stragglers can slow down the entire computation. To achieve the desired overall throughput, Swing must use devices collaboratively, taking into account their device heterogeneity.

Dynamism. In addition to device heterogeneity, the performance of the real-time sensing apps might be affected by external factors such as user mobility (captured by variations in signal strength), changes in applications running in the devices (captured by variations in CPU usage) and changes in the input data rate (captured by the queuing delay). To understand the impact of dynamism, we let A send video frames to B for processing, under three different scenarios: (1) B is placed in regions of different Wi-Fi signal strength; (2) B simultaneously runs another compute intensive benchmark task. (3) A sends frames to B at different rates.

As shown in Figure 2, Wi-Fi signal strength, processor utilization, and input data rate affect delays in transmission, processing and queuing. This suggests that Swing should dynamically (1) divert more frames to devices of strong signal strength, and (2) steer frames to accommodate the reduced computing capability when processor usage changes, and (3) control queuing delay on each device by matching its input data rates to its capability.

IV. Swing Overview

With the challenges in mind, we design and implement Swing, a general-purpose framework that enables the collaboration between multiple, mobile devices. In this section, we first present the overview of the system, including its programming model and workflow, and then detail its design and implementation.

A. Programming Model

Swing uses a dataflow programming model, that represents a mobile sensing app as a directed graph. Graph vertices correspond to computational parts of the app,

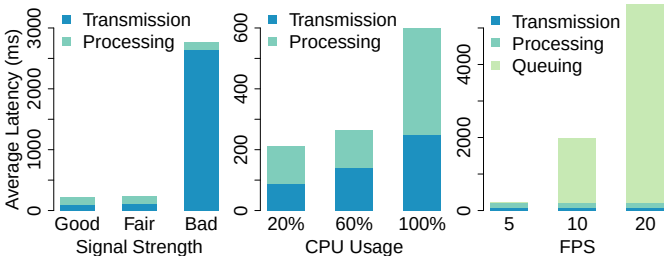

Fig. 2: Decomposition of delays in remote face-recognition processing. Wi-Fi signal strength primarily affects network transmission delay; CPU usage affects processing delay; input data rate affects queuing delay.

which we refer to as *function units*. App functionality is divided into multiple interconnected function units. For example, the face recognition app consists of four function units that (A) use the camera to capture video frames, (B) detect faces inside video frames, (C) match faces with names, and (D) display the results.

Graph edges represent data flow between function units. During app execution, each function unit receives a *data tuple* from a previous unit via an edge in the graph. The data tuple contains a list of serializable data structures, such as a bitmap image, a matrix of floating-point values or a text string. The function unit processes the incoming tuple, computes an intermediate result, encapsulates it in a tuple and passes it to the next unit in the graph. For a given function unit, a unit from which it receives data tuples is called an *upstream* unit, and a unit toward which it sends data tuples is called *downstream* unit. Each unit may interface with multiple upstream or downstream units. A unit without upstream is a *source* (e.g., A) and a unit without downstream is a *sink* (e.g., D).

To use the Swing framework, the programmer defines apps as function units with Swing APIs. Specifically, the programmer constructs the app graph by defining function units, including a source and sink, and defining edges that create a topology. For example, the code below describes the definition the face recognition app.

```
// The code below defines the application
    graph
public AppGraph compose(){
 /* Define tuple structure */
 ArrayList<String> tuple = new
     ArrayList<String>;
 tuple.add("value1"); //first part: a byte
     array
 tuple.add("value2"); //second part: a string
 /* Define function units */
 FunctionUnit src = FUBuilder(new
     Source(),srcId,tuple);
 FunctionUnit f1 = FUBuilder(new
     FunctionA(),aId,tuple);
 FunctionUnit snk = FUBuilder(new Sink(),
     snkId, tuple);
 /* Define topology */
 src.connectTo(f1);
 f1.connectTo(snk);
```

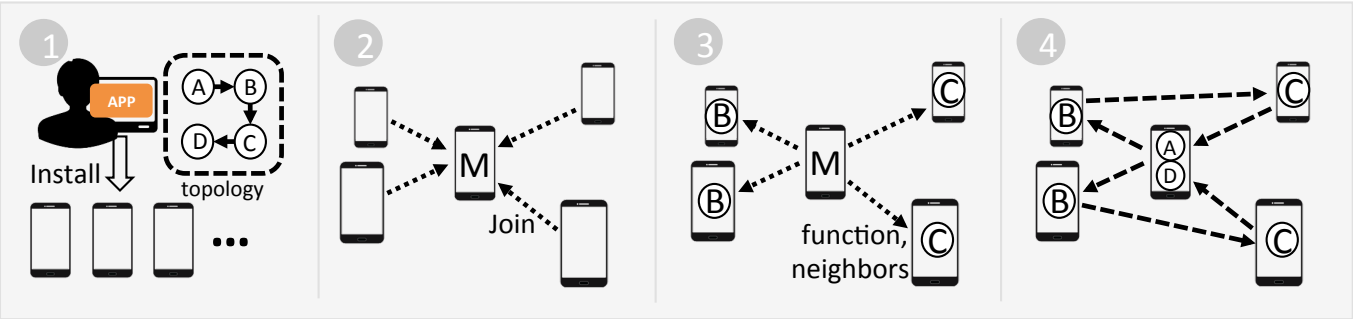

Fig. 3: Workflow of Swing Installing, Joining, Deploying and Running

```
}
```

Each function unit is programmed to first receive data, and then perform certain tasks. For example, the code below defines a function unit that transforms a received data tuple into a graphical object, processes it, and sends the result to the next tuple.

```
// The code below defines a function unit
public class FunctionA implements
    FunctionUnitAPI {
  @Override
  public void processData(Tuple data) {
    /* get a byte array from the data tuple
        received */
    // use "value1" as the key
    byte[] bytes =
        (byte[])data.getValue("value1");
    /* transform array to an image object */
    Mat mGray = new Mat();
    mGray.put(0,0,bytes);
    matToBitmap(mGray, mBitmap);
    /* process on the object */
    String name =
        faceRecognizer.recognize(mBitmap);
    /* pass the result data to the next
        function unit */
    Tuple output = data.setValues(null, name);
    send(output);
  }
}
```

The programmer can also define performance requirements that affect resource allocation and task scheduling, e.g., the maximum input data rate that needs to be sustained by an app. Swing enables programmers to express a single compute-intensive operation as separate function units, e.g., `detect()` and `recognize()`. This enables distributing computation load among multiple devices. At runtime, Swing determines their deployment with respect to devices' capabilities.

B. Workflow

Swing runs on devices connected through any IP network — for example, a WLAN. This WLAN can be an existing Wi-Fi network or can be created by one of the devices acting as Wi-Fi hotspot. When a group of devices intend to collaborate for a mobile sensing app (e.g. collaborative face recognition), they follow the workflow in Figure 3 to install the app, join the system, and execute the computation.

Step 1: Installing the App. Each device downloads and installs the specific stream processing app in her device. The app could be obtained from one of the devices or from an online app-store where developers submit their apps written with Swing APIs.

Step 2: Launching and Joining. Upon app installation, one device first launches a master thread, followed by others launching worker threads. The master initiates the app, broadcasts its IP address, launches a socket server and waits for connections. Workers then connect to the master's socket server.

Step 3: Deploying Function Units. The master deploys the app dataflow graph by assigning function units and connecting devices. Since each device has already installed all the function units, the master simply provides each worker the name of the function units it must activate . Then, the master informs workers about the IP addresses of their upstream and downstream workers, and the connections between workers can then be formed. The master thread is responsible only for control, bootstrapping connections and sending start/stop commands. It can co-locate on the same device with worker threads.

Step 4: Executing the App. After function deployment, the master instructs the worker devices with source function units to sense data and generate tuples. As source begin transmitting data tuples, downstream function units begin computation. In Figure 3④, source *A* distributes data tuples to the two devices running function unit *B*, which then pass intermediate results to downstream devices running function unit *C*.

In summary, developers write sensing apps using Swing APIs, and users download and install the apps on their devices. When a group of devices wish to run an app collaboratively, one of them initiates a master thread and the others initiate worker threads. The master then deploys the app's dataflow graph, which interconnects the worker threads across multiple devices.

C. Design and Implementation

Each Swing instance can be viewed as a distributed program that runs across multiple devices. A *master* thread

controls multiple *worker* threads and assigns software functions to them. Since masters and workers correspond to software threads, a single device could execute multiple worker and master threads.

We designed and implemented Swing on top of SEEP [9], a JAVA-based stream processing platform. SEEP provides a convenient interface for defining graph topologies by abstracting away the details of TCP socket connections and inter-thread communications. However, SEEP is targeted for static data center clusters as opposed to a dynamic mobile environment. The Swing architecture consists of the following service components.

Swarm Management Service. Swing dynamically manages device usages to achieve performance objectives and energy efficiency. Each upstream thread maintains a routing table with downstream threads' IDs and their weights, so that data tuples could be routed accordingly. We describe this algorithm in more detail in Section V.

Discovery Service. Swing automatically establishes connections between participating swarm devices. During initialization, the master broadcasts itself by registering a Network Service on the network, using Android Network Service Discovery (NSD). Each worker device maintains a background service that listens for the master and connects to it upon discovery.

Handling Joining and Leaving. In order to involve new devices as soon as they join, the master constantly listens for incoming connections and instantly activates function units on the new devices. The workers' routing tables are updated accordingly. When a network link is broken, due to poor wireless signal or a user leaving, the affected upstream units automatically remove the corresponding downstream from the routing tables and re-route data to other units. We evaluate dynamics in Section VI.

Serialization Service. Communicating through socket connections requires serialization. SEEP uses Kryo and Java serialization methods to serialize the data tuples transmitted between function units. However, mobile sensing apps may require transmitting customized objects, such as an image container, a multi-dimensional sensor vector, or a segment of audio stream. Swing extends SEEP's serialization function and transforms customized objects into a byte array, which is serializable, at the sender, and transforms the array back to the object at the receiver.

Reordering Service. Performance heterogeneity and dynamism cause each tuple's end-to-end delay to differ — tuples that are dispatched earlier may arrive later, and vice versa. To solve this problem, we buffer results as they arrive at the sink and sort them in-order before playback. A large buffer ensures better ordering but delays the display of the results. We will evaluate the effectiveness of our buffer sizing strategy in Section VI.

Background Service. Swing extends Android Service to run in the background without interrupting user activities. Swing acquires a CPU wake lock to prevent service termination due to processor sleep modes. Swing runs on Android version 4.0 or higher, which includes the majority of Android devices.

V. Resource management

In order to satisfy real-time requirements of mobile sensing apps, the primary objective of Swing is to minimize latency while meeting input rate requirements and using as few resources and energy as possible. This must be achieved through a low-complexity and distributed mechanism that rapidly reacts to mobile dynamics in real-time.

A. LRS algorithm

Swing uses a distributed low complexity routing algorithm that we call LRS (Latency-based Routing with worker Selection). LRS is executed at each upstream function unit in the application dataflow graph using information communicated periodically from its downstream function units (every 1 s in our implementation).

Each upstream function unit measures the total rate of its incoming data tuples Λ and estimates the mean latency L_i of each downstream function unit i. Based on this information, LRS operates in two steps outlined below.

Worker Selection. First, the upstream function unit selects a subset S of its downstream function units D. More specifically, it sorts function units in descending order of service rates $\mu_i = 1/L_i$ and selects the minimum number of function units S such that $\sum_{i=1}^{S} \mu_i \geq \Lambda$.

Sorting seeks to select the fastest downstream function units in order to avoid stragglers. Selecting the minimum number of function units seeks to minimize the amount of compute resources and energy consumed. The sum rate constraint seeks to satisfy the input rate requirement. If the sum rate constraint cannot be satisfied, all downstream function units are selected.

Data Routing. This step decides what fraction of incoming tuples will be routed to selected function units in S. Each upstream function unit maintains a routing table, which contains ID and a normalized weight p_i for all its downstream function units in D. p_i is computed based on the measured downstream latencies: $p_i = \frac{1/L_i}{\sum_j (1/L_j)}$. LRS routes incoming tuples to selected function units in proportion to normalized service rate p_i. Routing more data to faster downstreams avoids high queuing delays and straggler effects on slower downstreams. In addition, this assignment equalizes latencies among downstreams which helps minimizing maximum latency and minimize additional delays in case of tuple reordering requirements at the sink.

LRS uses probabilistic routing by treating the normalized weights as probabilities. Upon arrival of a data tuple, the upstream generates a weighted random number and sends the tuple to the specified downstream ID. This approach yields fast low complexity routing decisions per tuple as it only requires random number generation.

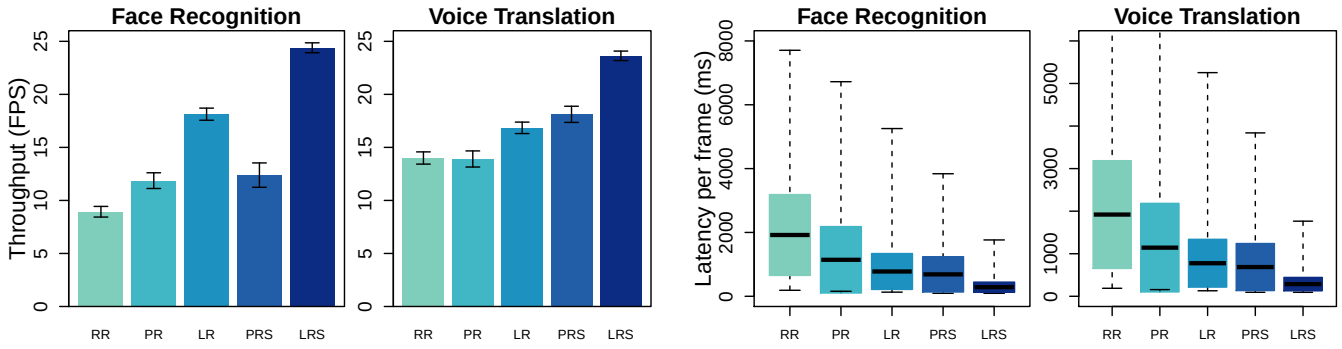

Fig. 4: Average system throughput and the min, max, average and variance of per-frame latency in two applications.

B. Latency estimation

L_i is estimated as a moving average of latency estimates obtained as follows. The upstream attaches a timestamp to each tuple. Each downstream, after processing the tuple, sends back an ACK with the original timestamp. Upon receiving the ACK, the upstream calculates latency estimate for this tuple by subtracting the timestamp from the current time. Thus, L_i includes network delay between the upstream and downstream device (dominated by transmission time), queuing delay and processing delay the downstream function unit, and the ACK transmission delay (negligible due to ACK's small size). In order to estimate L_i of the function units that were not selected in previous rounds, each upstream function unit switches periodically every few rounds to round robin mode for a short time, sending a few tuples to all its downstream function units.

C. Discussion

In summary, the key design points in LRS are Worker Selection and Latency-based Routing.

Worker Selection serves a dual purpose. First, it addresses the problem of stragglers by filtering out downstream function units running on devices that are slow or are on slow network links. Second, it aims to improve energy efficiency by using enough resources to meet an input rate target. However, Worker Selection may not be necessary if devices have similar capabilities. Therefore, one alternative is not to include Worker Selection and let data routing handle device heterogeneity.

Latency-based Routing aims to minimize the overall latency and avoids downstream function units on slow wireless links. However, it might rely heavily on function units running on devices with fast network links but computationally less capable. Some of the more computationally capable devices are newer and more energy-efficient. Therefore, one alternative to Latency-based Routing is Processing-delay-based Routing, which routes data to devices based only on their computational capabilities and potentially providing a higher energy efficiency.

In the upcoming evaluation section, we will compare the performance of LRS against the above alternatives.

VI. Evaluation

We evaluate Swing, assessing its ability to address key challenges in mobile swarm-based computing. We first explain the experiment applications, followed by the evaluations of comparing swarm management strategies. We then evaluate how the system adapts to user mobility.

A. Experiment Applications

We use two open-source sensing applications to evaluate Swing. The first app is face recognition [4], which uses the OpenCV CascadeClassifier class to detect faces in an image and the FaceRecognizer class to recognize faces. We modified its source code with Swing API to create four function units: reading video frames from files (source), detecting faces from frames (detector), matching faces with databases and return results (recognizer), and displaying results (sink). The size of each video frame is 400×226 pixels (6.0kB).

The second app is voice translation. It contains four function units: reading audio frames from files (source); recognizing audio streams into English words (based on CMU Pocketsphinx [18]); translating those words into Spanish (based on Apertium [15]); and displaying results (sink). The size of each audio frame is 72.0kB.

B. Comparison of Data Routing Methods

In this subsection, we compare the performance of **LRS** with four alternative algorithms, with respect to throughput, latency and energy efficiency metrics.

We deploy nine phones ($A \cdots I$ in Section III) and configure device A to act as source and run a master thread while the rest run worker threads. To account for network heterogeneity, we place devices B, C, D at locations of poor Wi-Fi signals, yielding slower wireless links.

We compare LRS with four different strategies:

(1) **RR** (round-robin routing). We let each upstream thread send data to all its downstream threads in turns, each tuple at a time. This is the default data distribution mechanism used in existing data center-based stream processing systems [9], [2], [3] and it is also used in recent mobile ones [23], [32].

(2) **PR** (Processing-delay-based routing w/o worker selection). This routing policy uses only processing delay W

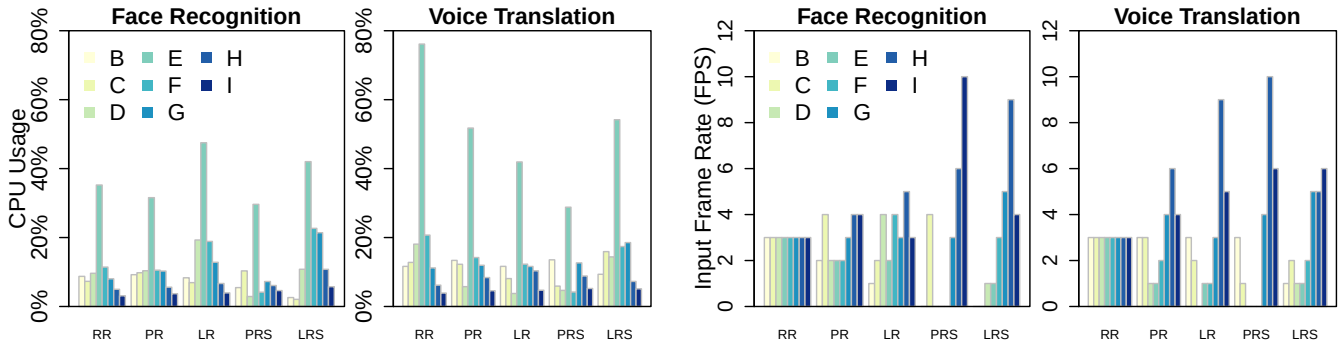

Fig. 5: Resource usage and input data rate of each device in two applications. RR sends an equal amount of data to each device. Swing consumes a larger share of processor time when the processor is weak (e.g., phone *E*) and a smaller share when it is strong (e.g., *I*). In contrast, LRS minimizes the usage of devices located in regions with weak signals (*B*, *C*, *D*) and devices that produce latency stragglers (*E*, *F*).

instead of latency *L*. It seeks to route traffic to the most capable and energy-efficient devices.

(3) **LR** (latency based routing w/o worker selection). Similar to LRS, but eliminating Worker Selection. The comparison between LR and LRS would show the impact of device heterogeneity, and how Worker Selection helps improve energy efficiency.

(4) **PRS** (processing-delay-based routing with worker selection). Similar to PR, but using Worker Selection.

1) ***Performance****:* Figure 4 shows the average throughput of the system and the min, max, average and variance of latency per frame. Latency based routing methods (LR and LRS) provide lower delay per frame (e.g., smaller mean and variance). Processing-delay based methods (PR and PRS) fail to provide the target rate of 24 FPS, because they schedule data purely based on the capabilities of downstream threads, regardless of their locations in the network. Specifically in this experiment, PR and PRS often route data to threads on device *B* and *C* which are located in locations of weak signals. As a result, the TCP and Wi-Fi rate adaptation protocols require the sender to lower network transmission rates for the devices of downstream function units, which directly reduces throughput and increases latency. The results also show that worker selection increases the throughput and decreases the average and variance of the latency. The reason is that worker selection filters out stragglers.

In summary, Swing's LRS strategy performs best in terms of throughput and latency. Compared with the baseline RR, LRS provides 2.7x improvement in throughput and 6.7x reduction in average latency.

2) ***Overheads****:* Swing incurs two types of overheads: processor utilization and data transmission. We evaluate these two overheads and estimate the energy consumption on each device.

Processor Utilization: To measure the processor utilization at run-time, all the devices are initially set idle, and when Swing launches, a background thread is launched simultaneously which executes `top` and records the measurements periodically.

Figure 5 shows average CPU utilization during the experiments. For different devices, the processor utilizations depend highly on their hardware capabilities. In the first two graphs in Figure 5, we observe that under RR, the same data arrival rate and computation load consume a larger share of processor time when the processor is weak (e.g., *E*) and a smaller share when the processor is strong (e.g., *I*). This is caused by the device heterogeneity.

For both apps, Swing incurs low CPU overhead — it distributes computations among devices with additional 14% CPU utilization per device, on average. This level of overhead has no perceptible impact on user experience. We note that some devices report CPU usage even when they are not selected for computation (e.g. D,E,F under PRS) — in addition to Swing, CPU utilization includes background operations from OS and other apps that we could not disable.

Data Transmission: Figure 5's right two graphs show the amount of data transmitted from the source to each worker device. The observations match the design of routing policies: RR evenly distributes, P* policies prefer faster devices (e.g. B,I), L* policies avoid devices in poor Wi-Fi locations (e.g. B,C), and *S policies select a subset of devices.

Power Consumption: Monitoring the actual real-time power consumption at app level for many different devices under various experimental configurations is extremely challenging [11], [33], [22]. Our experiments were not an exception. We thus use power consumption modeling approaches proposed by previous works [11], [33], [22]. These models were shown to yield adequate accuracy and can provide a good indication of the relative power performance of the different policies we evaluate. In order to create these models, we use the following profiling procedures.

Offline profiling: for each devices, we first measure its idle power. Then, we measure peak power by stressing the processor with 100% utilization for 30 minutes, recording the change in battery level in this duration, and estimating the corresponding change in energy given the device's battery capacity.

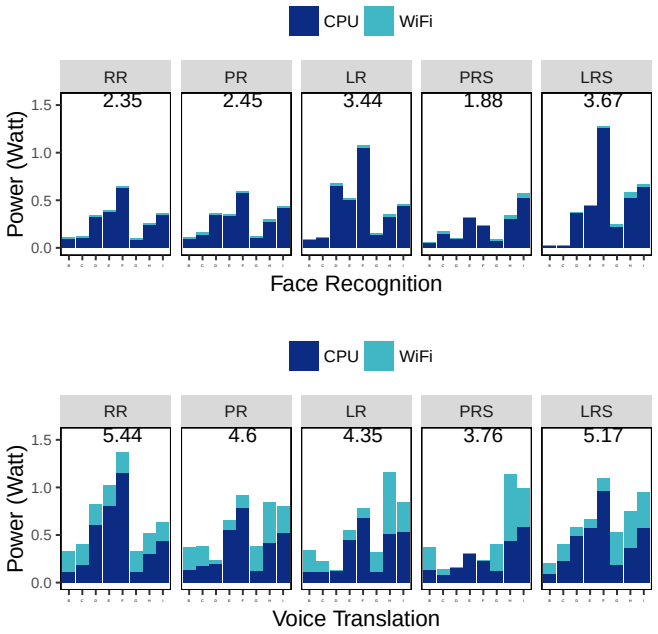

Fig. 6: Energy consumption each device.

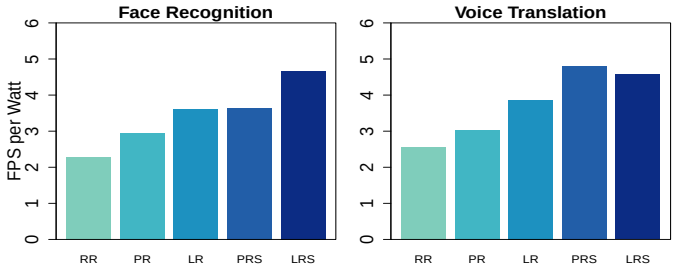

Fig. 7: Efficiency of routing schemes.

Online measurements: after building the profile model, during each experiment run, we measure the real-time CPU usage on each device. We then estimate the run-time processor power as a percentage of peak based on the measured processor utilization.

We estimate the Wi-Fi power using a similar method: measure idle power and measure peak Wi-Fi power by sending data at maximum bandwidth with `iperf`[30] for 30 minutes and recording the change in battery level. Given the profile model, we then estimate Wi-Fi power as a percentage of peak based on the measured data transmission rate during the experiments. Each bar in Figure 6 shows the estimated power consumption of CPU and Wi-Fi on each device, and the numbers on top indicate the aggregate power consumption across all the devices. CPU power consumption dominates Wi-Fi power consumption, which is indicative for the evaluated CPU-intensive apps. Slower devices (e.g., phone E) tend to consume more power due to the inefficiency of their processors. PRS consumes minimum power policy because it uses the fastest and more energy-efficient devices. On the other hand, LRS has highest power consumption as it may select devices with fast network connection but less energy-efficient processors.

Energy efficiency: In order to compare the policies in terms of both performance and energy consumption, we use FPS/Watt, an energy efficiency metric that reflects useful work done per Watt. To do so, we divide throughput (from Figure 4) by aggregate power consumption (from Figure 6). We observe that Worker Selection (*S) greatly improves energy efficiency. In addition, LRS outperforms all policies in the face recognition app and is slightly worse than PRS in the voice translation app. Since LRS is the only policy that can always meet the real-time input rate requirement, it is a preferable policy overall in terms of performance and energy efficiency.

Tuple Order: Tuples are sent out from the source in sequence, but performance and network heterogeneity will cause tuples to arrive in different order. The loss of tuple order leads to an unpleasant user experience, such as choppy video playback in the case of a face recognition app. We engineer the size of the buffer with respect to the data rate generated at the source, i.e., timespan of 1 second. Figure 8 illustrates this loss of order by showing the timestamp at which the result for each tuple arrives at the sink (see the gray dots) for the face recognition app. A perfectly ordered tuple sequence should produce a linear sequence of gray dots. We observe that the dots are scattered except in LRS. The solid lines in Figure 8 illustrate the playback times for re-ordered tuples when the reorder buffer of length 24 (i.e., one second) is applied. Routing methods with Worker Selection, especially LRS, have smoother curves than others because they use less devices and produce smaller variances in latency. LRS is better than PRS, as it reduces latency variances by taking into account the network transmission and the queuing delays in addition to processing delays.

C. Handling Mobility

In this subsection, we demonstrate how Swing adapts to mobile users, who join the system during computation, move in ways that affect network connectivity, and leave the system abruptly.

Joining: We start with phone A running the master and phones B, D running worker threads. As they compute, we launch Swing on G, which then joins the computation automatically. Upon detecting an incoming connection, the master connects G with B and D, and the later two add G's thread IDs to their routing tables to re-calculate all the probabilities (using LRS). Figure 9 (left) shows that, within a second of G's arrival, throughput rises to its maximum level of 24 FPS in the face recognition app. The system preserves all the existing links during the transition and no data is lost.

Leaving: In this experiment, we first deploy three worker phones B, G, H with the LRS data routing scheme. In the midst of computation, we manually terminate the Swing thread on G. Upon detecting the lost connection, the upstream traces the connection's downstream ID, removes

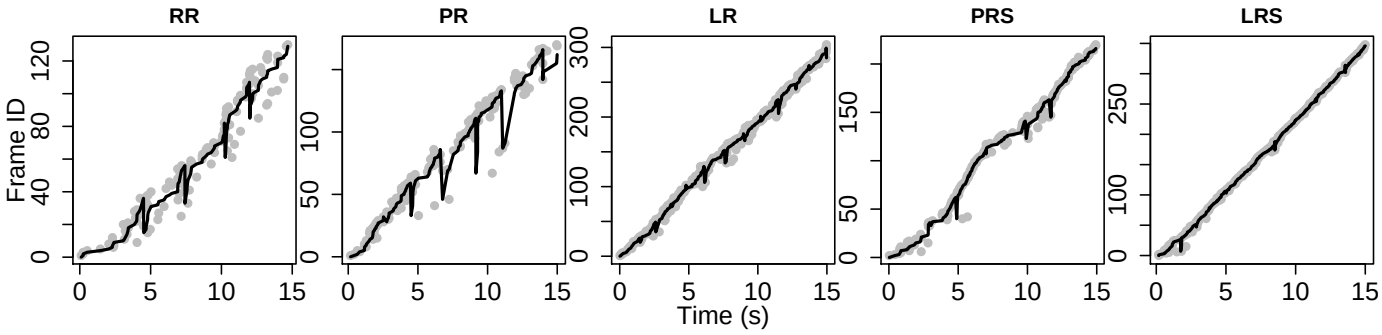

Fig. 8: Ordering of frames: gray dots represent frames' arrival timings; solid line represents the reordering using a buffer.

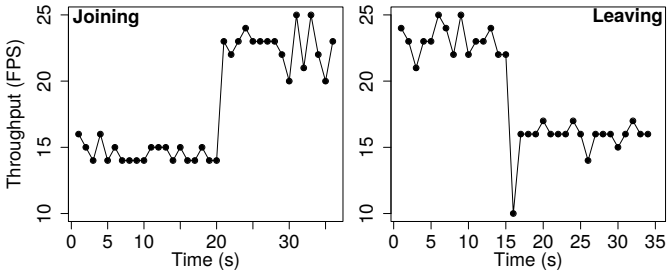

Fig. 9: Throughput changes when device joins, leaves

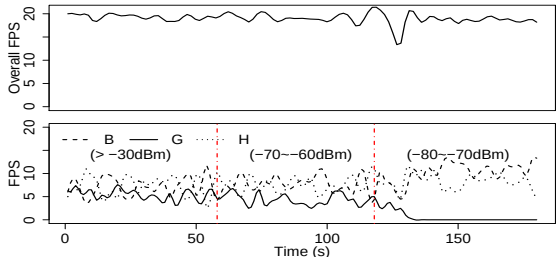

Fig. 10: Throughput, load changes when device moves

the ID from the routing table, and re-calculates data routing probabilities for the remaining downstreams. Figure 9 indicates that real-time throughput drops drastically after the device leaves and before the system updates the routing table. The upstream attempts to route data to the disconnected device and, during the recovery phase, 13 frames are lost. Yet, within one second, throughput recovers to 16 FPS, which is the best that can be achieved with the remaining devices.

Moving: The system deploys phones B, G, H with the LRS routing method, which copes with variations in network delay. Initially, phones are all placed at a location with good Wi-Fi signals (RSSI>-30dBm). After one minute, G's user walks to a location with slightly weaker signals (between -70dBm and -60dBm), stays there for one minute, and then walks to a third location with poor signals (between -80dBm and -70dBm). Figure 10 shows the effect of mobility on signal strength, which affects overall throughput (top graph) and per-device throughput (bottom graph). We observe that the overall throughput recovers quickly after G moves to a region with weak signals as Swing re-routes data to the other two phones.

VII. Conclusion

In this paper, we design, implement and evaluate a distributed mobile computing framework, targeting continuous sensing applications that are compute-intensive and delay-sensitive. We developed Swing, a framework that allows multiple mobile devices to perform computations based on a dataflow graph. We identified the major challenges for achieving the performance potential and propose resource management and routing techniques for coping with device heterogeneity and dynamics. We built a system prototype using Android devices and a Java-based stream processing platform. By evaluating with two sensing applications and multiple mobile devices on a wireless testbed, we show that performance requirements and efficiency can be satisfied through device management with minimal computational overhead.

Acknowledgment

This research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement Number W911NF-06-3-0001, W911NF-16-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon. This work was also supported by the National Science Foundation under grants CCF-1149252 (CAREER), CCF-1337215 (XPS- CLCCA), SHF-1527610, and AF-1408784. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these sponsors.

References

[1] Apache heron. https://twitter.github.io/heron/.
[2] Apache storm. https://storm.apache.org/.
[3] Ibm streams. https://www.ibm.com/cloud/streaming-analytics.
[4] Opensource face recognition application. https://github.com/ayuso2013/face-recognition.

[5] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[6] M. Y. Arslan, I. Singh, S. Singh, H. V. Madhyastha, K. Sundaresan, and S. V. Krishnamurthy. Computing while charging: Building a distributed computing infrastructure using smartphones. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 193–204, New York, NY, USA, 2012. ACM.

[7] Microsoft Azure IoT Edge. https://azure.microsoft.com/en-us/services/iot-edge, 2017.

[8] N. Banerjee, M. Corner, and B. Levine. An energy-efficient architecture for dtn throwboxes. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 776–784, May 2007.

[9] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 725–736, New York, NY, USA, 2013. ACM.

[10] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM.

[11] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.

[12] Z. Dong, L. Kong, P. Cheng, L. He, Y. Gu, L. Fang, T. Zhu, and C. Liu. Repc: Reliable and efficient participatory computing for mobile devices. In *Sensing, Communication, and Networking (SECON), 2014 Eleventh Annual IEEE International Conference on*, pages 257–265, June 2014.

[13] A. Dou, V. Kalogeraki, D. Gunopulos, T. Mielikainen, and V. H. Tuulos. Misco: A mapreduce framework for mobile systems. In *Proceedings of the 3rd International Conference on PErvasive Technologies Related to Assistive Environments*, PETRA '10, pages 32:1–32:8, New York, NY, USA, 2010. ACM.

[14] U. Drolia, K. Guo, J. tan, R. Gandhi, and P. Narasimhan. Cachier: Edge-caching for recognition applications. In *IEEE International Conference on Distributed Computing Systems (ICDCS), 2017*, ICDCS '17, pages 276 – 286, New York, NY, USA, 2017. IEEE.

[15] M. Forcada, M. Ginestí-Rosell, J. Nordfalk, J. O'Regan, S. Ortiz-Rojas, J. Pérez-Ortiz, F. Sánchez-Martínez, G. Ramírez-Sánchez, and F. Tyers. Apertium: a free/open-source platform for rule-based machine translation. *Machine Translation*, 25(2):127–144, 2011.

[16] Amazon AWS Greengrass. https://aws.amazon.com/greengrass, 2017.

[17] S. Guo, M. Ghaderi, A. Seth, and S. Keshav. Opportunistic scheduling in ferry-based networks. In *In WNEPT*, 2006.

[18] D. Huggins-Daines, M. Kumar, A. Chan, A. Black, M. Ravishankar, and A. Rudnicky. Pocketsphinx: A free, real-time continuous speech recognition system for hand-held devices. In *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*, volume 1, pages I–I, May 2006.

[19] P. Hui, A. Chaintreau, R. Gass, J. Scott, J. Crowcroft, and C. Diot. Pocket switched networking: Challenges, feasibility and implementation issues. In *Proceedings of the Second International IFIP Conference on Autonomic Communication*, WAC'05, pages 1–12, Berlin, Heidelberg, 2006. Springer-Verlag.

[20] Y. Kao, B. Krishnamachari, M. Ra, and F. Bai. Hermes: Latency optimal task assignment for resource-constrained mobile computing. *IEEE Transactions on Mobile Computing*, 16(11):3056–3069, Nov. 2017.

[21] E. Marinelli. Hyrax: Cloud computing on mobile devices using mapreduce. Master's thesis, Carnegie Mellon University, Sep 2009.

[22] R. Mittal, A. Kansal, and R. Chandra. Empowering developers to estimate app energy consumption. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Mobicom '12, pages 317–328, New York, NY, USA, 2012. ACM.

[23] Q. Ning, C. A. Chen, R. Stoleru, and C. Chen. Mobile storm: Distributed real-time stream processing for mobile clouds. In *2015 IEEE 4th International Conference on Cloud Networking (CloudNet)*, pages 139–145, Oct 2015.

[24] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM.

[25] M.-R. Ra, B. Liu, T. F. La Porta, and R. Govindan. Medusa: A programming framework for crowd-sensing applications. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 337–350, New York, NY, USA, 2012. ACM.

[26] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: Enabling interactive perception applications on mobile devices. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 43–56, New York, NY, USA, 2011. ACM.

[27] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, Oct. 2009.

[28] C. Shi, V. Lakafosis, M. H. Ammar, and E. W. Zegura. Serendipity: Enabling remote computing among intermittently connected mobile devices. In *Proceedings of the Thirteenth ACM International Symposium on Mobile Ad Hoc Networking and Computing*, MobiHoc '12, pages 145–154, New York, NY, USA, 2012. ACM.

[29] S. Teerapittayanon, B. McDanel, and H. Kung. Distributed deep neural networks over the cloud, the edge and end devices. In *IEEE International Conference on Distributed Computing Systems (ICDCS), 2017*, ICDCS '17, pages 328 – 339, New York, NY, USA, 2017. IEEE.

[30] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. iperf: Tcp/udp bandwidth measurement tool. 01 2005.

[31] T. Verbelen, P. Simoens, F. De Turck, and B. Dhoedt. Cloudlets: Bringing the cloud to the mobile user. In *Proceedings of the Third ACM Workshop on Mobile Cloud Computing and Services*, MCS '12, pages 29–36, New York, NY, USA, 2012. ACM.

[32] H. Wang and L.-S. Peh. Mobistreams: A reliable distributed stream processing system for mobile devices. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 51–60, Washington, DC, USA, 2014. IEEE Computer Society.

[33] F. Xu, Y. Liu, Q. Li, and Y. Zhang. V-edge: Fast self-constructive power modeling of smartphones based on battery voltage dynamics. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 43–56, Berkeley, CA, USA, 2013. USENIX Association.

[34] F. Yang, Z. Qian, X. Chen, I. Beschastnikh, L. Zhuang, L. Zhou, and J. Shen. Sonora: A platform for continuous mobile-cloud computing, 2011.

[35] N. Yigitbasi, L. Mummert, P. Pillai, and D. Epema. Incremental placement of interactive perception applications. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, pages 123–134, New York, NY, USA, 2011. ACM.

[36] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.