Demo: A Framework for Collaborative Sensing and Processing of Mobile Data Streams

Songchun Fan
Duke University

Theodoros Salonidis
IBM T.J. Watson Research

Benjamin Lee
Duke University

1. INTRODUCTION

Emerging mobile applications involve continuous sensing and complex computations on sensed data streams. Examples include cognitive apps (e.g., speech recognition, natural language translation, as well as face, object, or gesture detection and recognition) and anticipatory apps that proactively track and provide services when needed. Unfortunately, today's mobile devices cannot keep pace with such apps, despite advances in hardware capability. 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 [2, 3, 4]. However, cloudlets may not be widely deployed and access to cloud infrastructure may yield high network delays and can be intermittent due to mobility. Morever, users might hesitate to upload private sensing data to the cloud or cloudlet. A second approach offloads to accelerators by rewriting code to use DSP or GPU within mobile devices. However, using accelerators requires substantial programming effort and produces varied benefits for diverse codes on heterogeneous 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 users share a common sensing and computation goal. For example, travelers could benefit from real-time translation of native speakers using collaborative processing on their mobile devices.

This demo presents Swing (SWarm computing for mobile sensING), a distributed framework for compute- and sensing-intensive mobile apps. Swing views a collection of nearby mobile devices as a *swarm* and aggregates them to efficiently compute a shared answer. This is achieved by effectively managing stream processing parallelism, mobility, and device heterogeneity. Swing uses an intuitive programming model where mobile sensing apps are expressed by a dataflow graph.

MobiCom'16 October 03-07, 2016, New York City, NY, USA

ACM ISBN 978-1-4503-4226-1/16/10.

DOI: http://dx.doi.org/10.1145/2973750.2985620

2. Swing FRAMEWORK

Programming Model. Swing uses a dataflow programming model. A Swing app is represented by a directed graph. Graph vertices correspond to computational parts of the app (which we call *function units*) and graph edges represent data flow between function units. During app execution, each function unit receives *data tuples* from an *upstream* function unit specified by a directed edge in the graph. Each tuple contains a list of data structures. The function unit processes each incoming tuple, computes a result, encapsulates the result in another tuple, and passes it to the next *downstream* unit in the dataflow graph.

Developers write Swing apps using Java-based APIs. The APIs consist of defining data tuples and constructing the app graph topology by defining function units and edges between them. Swing enables expressing a compute-intensive operation as separate function units. This enables reduction of per-device computation through executing different function units on different devices. For example, a face recognition app can be represented by a line graph of four sequential function units: *source* using the camera to capture video frames, *detector* detecting faces inside video frames, *recognizer* matching faces with names, and *sink* displaying results. These function units are deployed to all devices as part of the app. During execution, Swing's resource management framework decides in real time which function units will execute at each device.

Workflow. The following steps are executed when a group of users wish to run a Swing app.

Step 1: Installing the App. Each device downloads and installs the app, obtained from one of the devices or from online appstore where developers submit apps developed with Swing APIs.

Step 2: Launching and Joining. One device initiates the workflow by launching a master thread, followed by others launching worker threads and connecting to the master.

Step 3: Deploying Function Units. The master deploys the app graph by providing each worker 1) the names of the function units it must activate, and 2) the IP addresses of their upstream and downstream workers to form the connections.

Step 4: Executing the App. 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.

Design and Implementation. Swing is implemented atop SEEP [1], a JAVA-based stream processing platform. SEEP provides an interface for defining graph topologies by abstracting the details of TCP socket connections and inter-thread communications. It also provides a parallel processing interface for scaling up or down computation threads. SEEP targets static homogeneous data center environments. In contrast, Swing

Figure 1: The demo setup includes multiple Android devices and a Wi-Fi router. One device acts as master and rest act as workers.

```
//Define the topology graph
public QueryPlan compose(){
 ...
 FunctionUnit src = FUBuilder(new Source(),srcId,tuple);
 FunctionUnit f1 = FUBuilder(new FunctionA(),aId,tuple);
 FunctionUnit f2 = FUBuilder(new FunctionB(),bId,tuple);
 ...
 src.connectTo(f1);
 f1.connectTo(f2);
 ...
}
//Define function unit B
public class FunctionB implements FunctionUnitAPI {
 ...
  @Override
  public void processData(Tuple data) {
    //get data from previous unit
    byte[] bytes = (byte[])data.getValue("value1");
    ...
    Mat mGray = new Mat();
    //transform received data to a graphical object
    mGray.put(0,0,bytes);
    //process on the data
    ...
    //pass the result data to the next function unit
    Tuple output = data.setValues(resultBytes);
    send(output);
  }
```

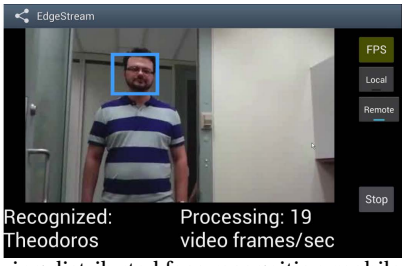

Figure 2: Examples of Swing distributed face recognition mobile app code (left) and screenshot (right).

provides mechanisms to handle heterogeneous and dynamic mobile environments.

Each Swing app is a distributed program of software threads running on multiple devices. A *master* thread controls multiple *worker* threads and assigns function units to them. Swing optimizes performance through dynamic data routing and function activation among devices. Each Swing function unit is implemented as a separate thread. Each upstream thread maintains a routing table with downstream threads' IDs and their weights, so that data tuples could be routed accordingly.

Swing features various novel data routing algorithms which can exploit parallelism and distribute load to optimize for various objectives along the throughput, delay and energy space. These algorithms are low-complexity, fully distributed and can react to mobile dynamics in real-time. This is achieved with lightweight monitoring mechanisms where downstream threads report delay or energy measurements to upstream threads. Using this information, upstream threads assign weights to their downstream threads and route their incoming data streams according to these weights. Each optimization objective corresponds to a different calculation of the routing weights.

Swing can run on devices connected through an IP network. In this demo we use a WLAN-based implementation. Swing handles new device arrivals and disconnections due to device mobility or power-down. During initialization, the master broadcasts its address by registering a Network Service, using Android Network Service Discovery (NSD). The master constantly monitors the WLAN for incoming devices. When a new device joins, the master activates function units on it and signals existing workers to update their routing tables accordingly. Each device monitors the wireless links to its downstream devices. When a link disconnects, the upstream units remove the downstream units from the routing tables and re-route tuples to other units.

3. DEMO DESCRIPTION

Our demo setup consists of mobile devices connected to a Wi-Fi access point as shown in Figure 1. The demo will illustrate benefits of using proximal devices for performing compute intensive tasks. These benefits include augmenting the compute and data boundaries of a single device.

Swing APIs. The demo will first show to the attendees how mobile apps can be developed using Swing APIs. We will use four Swing apps as examples: face recognition, language translation, fall detection and image-based motion detection. An example code is shown in left part of Figure 2.

Distributed face recognition app. This part of the demo will use a face recognition app to demonstrate the capabilities of the Swing runtime environment. The first part will show that executing face recognition on a video using a single smartphone is very slow and does not meet real-time requirements. Also faces of the people in the video are detected but not recognized because the phone's face database is incomplete.

In the second part, the smartphone will discover and use proximal helper devices to process the same video. These devices will include Android devices in different forms such as smartphones, tablets, or smartwatches. The face databases of the helper devices will collectively contain the faces of the people in the video.

The attendees will be able to see two effects in real time. First, a high computation speedup achieving video processing in real time. Second, in addition to detection the names of the people will now appear on the processed video since the face databases of helper phones will be utilized. The attendees will be able to interact with the demo in various ways. At start time, they will be able to modify the input load (through video frame rate) and the data routing policy. They will be able to observe performance by inspecting the quality of the analyzed video or a dashboard which will show system throughput, delay and energy consumption in real time.

Cloud infrastructure mode. We will also demonstrate that Swing can easily integrate with cloud infrastructure using Android VMs. More specifically we will use an Android VM on a laptop connected to the same Wi-Fi AP as the devices and will demonstrate performance for cloudlet and cloud scenarios by varying the network delay to the Android VM using dummynet. In this way, the attendees will be able to see how processing in cloud, cloudlet and proximal devices relate to each other.

4. DEMO REQUIREMENTS

The demo will use Android devices (smartphones, tablets, smartwatch), a Wi-Fi access point, a monitor and a laptop. At least two power plugs and a table large enough to place the above equipment will be needed. Setup time will be minimal. Internet access would be desirable although not a requirement for this demo.

5. REFERENCES

[1] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. SIGMOD '13.

[2] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: Elastic execution between mobile device and cloud. EuroSys '11.

[3] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: Making smartphones last longer with code offload. MobiSys '10.

[4] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: Enabling interactive perception applications on mobile devices. MobiSys '11.