PMDC: Programmable Mobile Device Clouds for Convenient and Efficient Service Provisioning

Zheng "Jason" Song and Eli Tilevich
Software Innovations Lab, Virginia Tech, VA
Email: {songz, tilevich}@cs.vt.edu

***Abstract*—Modern mobile devices feature ever increasing computational, sensory, and network resources, which can be shared to execute tasks on behalf of nearby devices. Mobile device clouds (MDCs) facilitate such distributed execution by exposing the collective resources of a set of nearby mobile devices through a unified programming interface. However, the true potential of MDCs remains untapped, as they fail to provide practical programming support for developers to execute distributed functionalities. To address this problem, we introduce a microservice-based *Programmable MDC* architecture (PMDC), highly customized for the unique features of MDC environments. PMDC conveniently provisions functionalities as microservices, which are deployed on MDC devices on demand. PMDC features a novel domain specific language that provides abstractions for concisely expressing fine-grained control over the procedures of device capability sharing and microservice execution. Furthermore, PMDC introduces a new system component—the microservice gateway, which reconciles the supply of available device capabilities and the demand for microservice execution to distribute microservices within an MDC. Our evaluation shows that MDCs, expressed by developers through the PMDC declarative programming interface, exhibit low energy consumption and high performance.**

I. Introduction

Mobile device users are continuously increasing their expectations on the functionality and quality of service of mobile applications. Meeting these expectations requires making use of sensory data, multimedia, and artificial intelligence algorithms. Unfortunately, applications that incorporate these features tend to require inordinate amounts of computational power, storage, battery budgets, high network throughput capacities, and extensive utilization of sensory resources. We observe that a typical modern mobile device is almost always operated in the vicinity of other mobile devices, many of which belong to groups of trusted or semi-trusted users, such as households and project teams. Mutual sharing of resources across co-located devices offers opportunities to create novel mobile apps and improve the quality of services of existing apps.

Leveraging the resource capabilities (computation, storage, sensing, network, etc.) of such co-located mobile devices at the edge of the network to execute tasks is generally referred to as *Mobile Device Computing (MDC)* [1]. MDC has been widely adopted in multiple usage scenarios: 1) using the computational resources to perform tasks, including speech, image recognition [2], [3], [4], [5], [6], [1], [7], [8], [9], [10]; 2) using the sensors, including GPS, motion sensor, microphone, and camera to collect sensory data for mobile sensing, localization, and video/audio generation [11], [12], [13], [14]; 3) using the network to optimize latency and throughput (one well-recognized use case is video streaming) [15], [16], [17]; 4) other use cases (e.g., using the storage for searching [18], using the phone speaker to tell a story [19], and using the touch screen for interactive gaming [20]).

However, lacking a coherent programming framework prevents developers from effectively leveraging MDC. Distributing and deploying the required functionalities to MDC for execution remains an unsolved problem. In traditional clouds, functionalities are expressed as pre-deployed cloud-based services, with centralized registry-based lookup strategies[21]. However, MDCs operate across a collection of nearby devices, an environment that changes constantly due to device mobility. Hence, finding a suitable device to execute a given functionality and delivering the functionality's executable code to the found device stand on the way of practical MDC applications.

Several prior works present strategies for implementing MDC applications (e.g., Serendipity[5], Mobile Fog[22], CoCam[11], ColPhone[19], and FemtoCloud[23]). Regarding the problem of finding the most suitable device to execute a functionality, these prior approaches either disregard the differences in device resource capabilities, or require low-level code to implement the communication logic for co-located mobile devices. Regarding the problem of deploying the required functionality on the found device, the prior approaches either transfer executable code across devices or require that the task executable files be pre-deployed.

In this paper, we introduce a novel system architecture, based on microservices. Although known for their applications in cloud-based scenarios [24], [25], microservices also fit naturally for the MDC environments. Microservice architectures express application functionality as a collection of interacting micro functionalities, each represented and managed as an external service. Similarly, our architecture represents and manages remote functionalities as microservices, which can be invoked on demand. Further, our architecture delivers the executable packages to the available MDC devices by downloading them from a trustworthy microservice market.

In particular, our software architecture facilitates the process of finding the most suitable device to execute a microservice. Programming support is provided via a domain-specific language that makes it straightforward to express: 1) capabilities offered by the available MDC devices, 2) microservice demands and their non-functional requirements (NFRs) (e.g., latency, reliability, cost, or any other microservice-specific aspects). We also notice that it would be impossible to directly translate device capabilities into NFR satisfiability, without the domain-specific knowledge possessed by microservice developers. Hence, the architecture features a novel network component, the microservice gateway, responsible for collecting device capabilities in order to estimate how they satisfy the NFRs.

The major contribution of this work is three-fold:

- A microservice-based software architecture that lowers the barrier for mobile app developers to use MDCs.
- A domain-specific language and its distributed runtime for expressing and matching the application's functionality demand and the MDC resource supply.
- A realistic use case implementation and performance evaluation of the aforementioned architecture.

In the rest of the paper, we start by analyzing the programming requirements and obstacles of MDC in Section II. Then, we present our system architecture design in Section III. Section IV introduces the domain specific language in details and Section V demonstrates the device selection procedure on the local gateway. Section VI describes our implementation and evaluation results. Section VII compares our approach with existing research and Section VIII concludes this paper.

II. Requirement Analysis & Solution Overview

The research literature motivates MDC with several typical use cases. We first analyze these cases to identify common obstacles in leveraging MDCs. Then, we briefly introduce our approach that removes these obstacles.

A. Programming Requirement

One typical MDC application scenario is facial recognition[23], depicted in Fig.1. A smartphone application needs to search for a given face from all photos in an album. Facial recognition is known to be both computationally intensive and energy consuming. Surrounding mobile devices can form an ad-hoc MDC to perform this functionality, splitting the work between the participating devices. The following discussion will continue referring to this scenario to explain our solution.

Another typical scenario is capturing and sharing images of a live concert from different view points [11]. While individual concert goers view the performance from a particular vantage point, they can enrich their experience by viewing the performance from a variety of different points, provided by other concert goers at various vantage points. This scenario requires reciprocity—a set of concert goers must agree to capture and share their respective views of the performance.

The last typical scenario is forming a device-to-device (D2D) network to accelerate data transmission [15]. Uploading a high-quality video can be time consuming, as the upload speed of a cellular network is typically lower than the download speed. An app needing to upload a media file fast can split it into pieces, so each piece can be uploaded by a nearby mobile device, thus multiplying the overall upload speed.

B. Analysis & Technical Obstacles

We analyze system design requirements from the developer's perspective. When a mobile application needs to request a nearby MDC to execute a task, the MDC has to allocate one or more devices for the execution. Cloud-based setups with fixed resource locations can consult centralized registries. However, this approach may not be suitable for MDC applications, in the presence of device mobility. Maintaining an MDC registry requires a localization procedure that is known to incur high energy consumption, as all participating devices need to periodically update their locations [11]. Another known strategy is having the MDC devices periodically announce their available functionalities via a D2D broadcast[26]. However, there is a great number and diversity of functionalities possessed and shareable by MDC devices. It would be inefficient to D2D broadcast all the available functionalities for a non-trivial number of devices.

Besides, when allocating MDC devices for a task, one must also consider the task's NFRs. Different mobile apps may require the same functionality, but with different NFRs. Consider the functionality of image capturing. This use case as presented in [11] requires the selected device to capture photo from a certain angel, while a surveillance app may require that the selected device persist the captured images to be retrieved at some point in the future. Notice that one cannot directly infer the NFRs from a device's available capabilities. For example, in the aforementioned image capturing and sharing use case, the viewpoints of MDC devices have to be calculated from their locations. Hence, MDC device allocation must consider both the device capabilities and the app's NFRs.

Finally, MDC devices should be able to provide the required functionality without introducing security vulnerabilities. For example, in real use cases, one cannot assume that the required

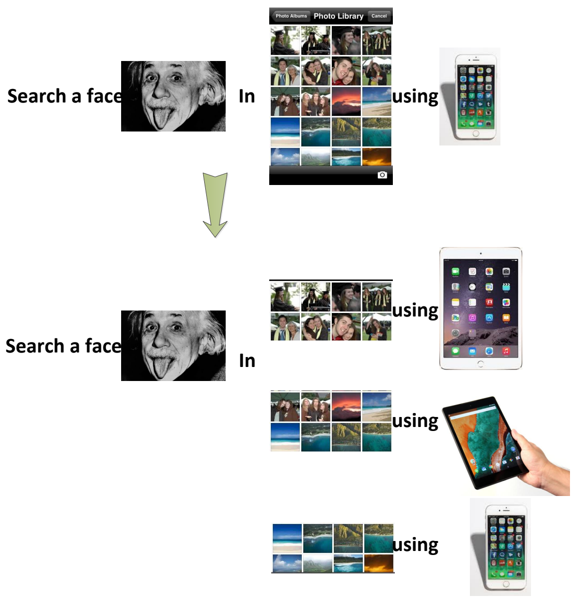


Figure 1: Usage Scenario 1: Face Recognition

execution package of a functionality be pre-deployed on MDC devices. When an MDC device is selected to perform facial recognition, the execution package containing facial recognition needs to be deployed on the device at runtime. The existing methods that transfer executable code between devices is vulnerable to attacks.

C. Solution Overview

Our software architecture solves the technical obstacles described above. The architecture structures applications as a collection of microservices—self-contained execution units, accessible by external clients through standard interfaces. The functionality demands are expressed as microservice requests, and carried out by microservice invocations.

III. System Architecture

Our software architecture is supported by the system runtime, which comprises four parts (see Fig.2): 1) a client device that requests a functionality from MDC; 2) a local device that serves as gateway by maintaining an up-to-date mappings between the available MDC devices and their capacities; 3) a microservice market (MSM for short), a cloud-based repository that delivers the executable code of a given microservice; 4) a set of MDC devices that share their capabilities, as detailed next.

A. MicroService Market (MSM)

MSM[27] combines features of application markets and service repositories. Following the application market model enables devices to automatically download and execute the required microservices, while following the service repository model enables application developers of the client apps to implement the required functionalities as microservice invocations, to be executed by MDC devices.

A microservice represents a certain functionality (e.g., getting temperature sensor readings, performing facial recognition algorithm on a given image). The microservice developers submit microservices to MSM, containing a unique identifier for service invocation, an NFR estimation package to be run by the gateway, and execution packages for key mobile platforms. To leverage such functionality, an application developer only needs to browse through the catalogs of microservices and invoke the microservice that provides the required functionality.

In the original design of MSM [27], a mobile device must download the microservices before it can be allocated to provide them. The devices are responsible for estimating their fitness to satisfy the NFRs of a given task and report the results to the gateway. By contrast, our new design enables the gateway to estimate how well the available devices can satisfy a tasks' NFRs, prior to deploying any microservices.

B. Local Gateways

A typical cloud-based microservice architecture features a centralized service registry, a collection of registered device-to-microservice mappings, with a remote interface through which clients can bind themselves to the microservices they want to invoke. Notice that MDC applications need to invoke microservices on the devices reachable via short-range communication methods (e.g., WiFi, Bluetooth), rendering cloud-based registries inapplicable.

Hence, our system architecture features a novel system component: a local gateway that replaces the standard cloud-based service registries. Each mobile device cloud should have a local gateway that could be either a stationary device, connected to a permanent power supply, or a battery-operated mobile device. Unlike its cloud-based counterparts, local gateways maintain a registry of available device capacities of the MDC, instead of the microservices provided by the devices.

C. Runtime Support on Devices

The runtime runs as a regular mobile app on the server and client devices. In general, the runtime accepts an MCL script to execute, either from the application via inter component communication (ICC), or from other devices via socket-based HTTP requests. On an MDC device, the programmer can specify the capability to share by interacting with the device's runtime using an MCL script. On a functionality demanding device, an app can first find the MDC device by querying the local gateway using an MCL script, and then invoke the microservice on the MDC device by passing it an MCL script with execution parameters.

D. Execution Flow

Fig.2 also introduces our system architecture's execution flow. The mobile devices periodically register their shared capabilities to the connected gateway (step 0). When a mobile app requires to execute a microservice on MDC, it first sends MCL scripts to the runtime on the client (step 1). The runtime then interacts with the reachable gateway in its vicinity, to query the most suitable device for microservice execution (step 2). The gateway downloads the NFR estimation algorithm of the required microservice from MSM (step 3), applies it to select the most suitable MDC device(s), and sends the connectivity information of the selected devices back to the client. Then, the client connects to the selected device to initialize the microservice execution (step 4). The selected device downloads the execution package from MSM, and sends the execution results back to the client (step 5).

IV. MCL Definition and Use Case

In this section, we first introduce the grammar of MCL, and explain its semantics for expressing the supply of device capabilities and the demand for microservices.

A. Functional Requirement

We first summarize what functions MCL provides:

1) Specify device capability to share: The MDC devices need to specify what capabilities to share.
2) Find device for executing a microservice: The functionality demanding device needs to obtain one or more MDC devices, whose capabilities 1) fulfill the general execution requirements of a microservice (e.g., in use

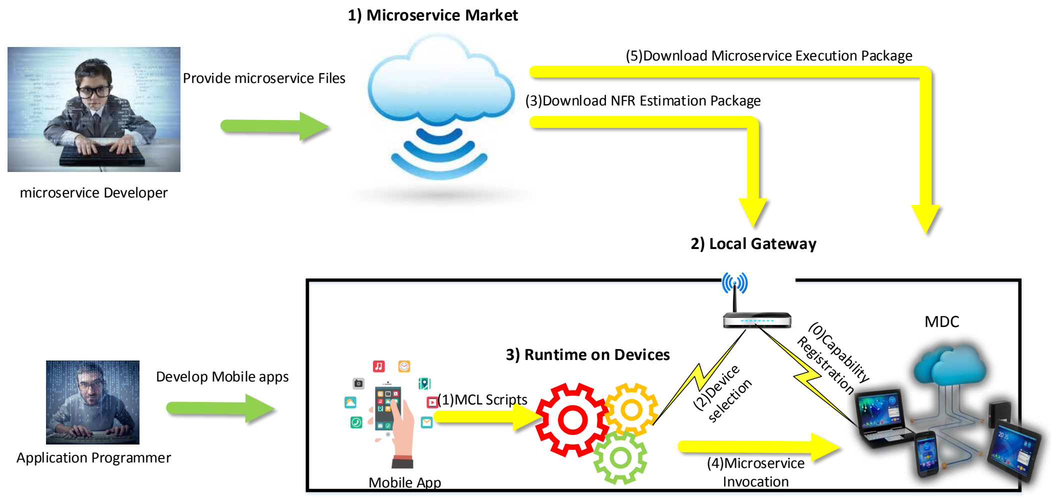


Figure 2: System Architecture Overview.

```
<MCL Script> ::= <Action> <Target> <Parameters>
<Action> ::= "reg"|"stop"|"query"|"exec"
<Target> ::= {<Resource> ","}+ | <Microservice>
<Resource> ::= "network"|"compute"|"sensor/"<Sensor>
  <Sensor> ::= "GPS"|"Cam"|"Mic"|"Motion"|"Light"|String
<Microservice> ::= String
<Parameters> ::= <Lease>|<Device Selection>|<Execution Param>
  <Lease> ::= "-t=" Numeric "-c=" Numeric
  <Device Selection> ::= ["-n=" Numeric]["-h="String]["-l="String]
  <Execution Param> ::= [String "=" String|Numeric]+
```

Figure 3: MCL EBNF Definition.

case 2, taking picture requires the device to share camera), and 2) best satisfy the NFRs (e.g., in use case 1, the app prefers an MDC device that can finish facial recognition most quickly).

3) Execute a microservice on a device: The functionality demanding device can start microservice execution on a selected MDC device.

B. Grammar Definition

An MCL script comprises three parts: `Action`, `Target`, and `Parameters`. `Action` stands for the method, which includes (1) register device capabilities, and remove the registered information (`reg`/`stop`), (2) query microservice provisioning (`query`), and (3) execute microservice (`exec`). The `Target` can be either `Resources` (for `reg` and `Stop`), or `Microservice` (for `query` and `Exec`). The `Resources` includes network, computing and sensors (including GPS, camera, microphone, motion sensors, light sensors, and etc.). `Microservice` is a string representing a unique ID of the related microservice function (e.g., "faceReco").

`Parameters` describes the action. When registering device capabilities, MCL enables specifying the leasing time (`-t`, for how long the capabilities will still be available), the incentive multiplier (`-c`, to be used to calculate the overall incentive for invoking microservice), and the device's status (e.g., CPU power, memory, CPU usage status, accuracy of sensors). When querying the device for microservice invocation, `Parameters` can be used to describe how many devices are requested (`-n`), as well as the NFRs(`-h=feature` indicates to select device with the highest value of `feature`,`-l` for the lowest). When executing a microservice, `Parameters` can be used to specify the runtime parameters to be bound to the microservice's execution.

C. Use Cases

Register/Stop Resources: An MDC device can register its device capability as available for remote execution, as well as stop such sharing. By leveraging such function, the programmers can decide what capabilities to share, based on the device owner's permission and the device's real-time status. The example program given in Fig. 4 shows two procedures: 1) reading the user's permission, and get all available device capabilities for remote execution (line 1-3); 2) specifying that when some computational intensive applications are running and the CPU load is high, stop sharing the compute capability for remote execution (line 4-5).

```
initialize registry
read user's permission and get available resource
reg.run("reg compute, sensor/Cam -t=1800")
if CPU.usage>50
   reg.run("stop compute")
```

Figure 4: MCL Example for Claiming Shared Capability.

Query and Execute Microservice: The functionality demanding devices can query for the most suitable MDC devices to execute a microservice, and request to execute the microservice on the selected device. The example program given in Fig. 5 shows how the motivating example 1 can be implemented in MCL. It also comprises two procedures: 1) query and get three devices for executing microservice "faceReco", with the highest estimation of the execution speed (line 4); 2) split all photos into three equal shares for the three devices, execute "facoReco" microservices for each photo (line 6).

```
initialize registry
read images: imgs = readDirectory("...");
separate into 3 shares: imgs_0, imgs_1, imgs_2
devices = reg.run("query faceReco -l=time -n=3");
for (IMAGE img : imgs_0) {
  devices.get(0).run("execute facoReco -img="+img);}
```

Figure 5: MCL Example for Executing Facial Recognition.

V. Device Selection Mechanism

When processing a microservice request, the local gateway first selects a device most suitable to service the request through the *device selection* procedure. The procedure matches between the requirements of executing a given microservice and the capabilities of the available devices.

Revisiting the facial recognition example: a gateway collects information about the available devices, including their CPU frequencies, memory sizes, and current workloads. Upon receiving a request to recognize a face in an image, the gateway consults the collected information to predict how well each device would satisfy the NFRs of the face recognition microservice (in this case, total execution time). However, predicting how fast a device can execute the facial recognition microservice is non-trivial: not only must the gateway be aware of the device's status, but it must also be able to determine how each aspect of that status would affect the total execution time, which is domain-specific knowledge possessed only by the developers of the face recognition microservice.

In our system design, it is the microservice developers who are expected to provide this domain-specific knowledge alongside the microservice itself. Specifically, microservices include an NFR estimation component. Local gateways download microservice packages from the MSM and execute their NFR estimators to select the most suitable device for the corresponding microservices. Next, we describe the device selection procedure in detail.

A. Web Interface on Local Gateways

The local gateway provides two web interfaces, for MDC devices to register their capabilities, and for microservice demanding devices to query for suitable server devices.

Fig. 6 demonstrates the interface for registering device capabilities. The `device status` currently includes CPU frequency, remaining energy status, memory usage, network speed, and sensor accuracy.

```
Interface 1: resourceRegistry
Parameters: resource = String
                 t = numeric
                 c = String
     {device status = numeric} +
Return: [Registration Success|Fail]
```

Figure 6: Capability Registration Interface

```
Interface 2: deviceSelection
Parameters: Microservice = String
                  n = numeric
                  h = String
                  l = String
Return: [Connection info of Devices|null]
```

Figure 7: Microservice Selection Interface

Fig. 7 demonstrates the interface for querying for suitable server devices. The client needs to provide a microservice ID, how many devices to select(`n`), and the NFRs (`h`/`l` for the highest/lowest estimated value).

B. Estimating NFR Satisfaction

Upon receiving a device selection request from a client, the gateway downloads the NFR estimation component of the required microservice from the MSM, and starts matching the device capability and execution requirements. Fig. 8 demonstrates an example of the NFR estimation package for microservice `faceReco`. Method `isCapable` checks whether a device is capable of executing a given microservice, and methods `energy` and `time` estimate how a device would satisfy these two NFRs, respectively.

```
class FaceRecoEstimator(val d: Device)
                  extends Estimator {

 override def isCapable(): Boolean =
           { d.compute().available() }

 def energy(): Int = 100 - d.battery.toInt

 def time(): Int = {
  var ret: Int = d.CPU * (1 - d.CPUusage)
  if (d.memory > 2000) ret *= 2
  ret
 }}
```

Figure 8: Estimating NFR Satisfaction (in Scala)

Revisit the device selection request expressed in MCL script, as shown in Fig.5. Upon receiving the request, the gateway first finds a set of nearby devices, whose `isCapable` methods return true. Then, it executes the `time` method on each device, selects the three devices with the lowest expected execution times, and returns the information to the requester about how to connect to these three devices.

VI. Reference Implementation and Evaluation

In this section, we report on 1) the reference implementation of the described architecture; 2) the performance of the implementation; 3) the comparison between our device selection procedure and that of key other designs. We implement the local gateway on a off-the-shelf WiFi router, a generally available infrastructure component, thus indicating the wide applicability of our system design.

A. *Implementation Specifics*

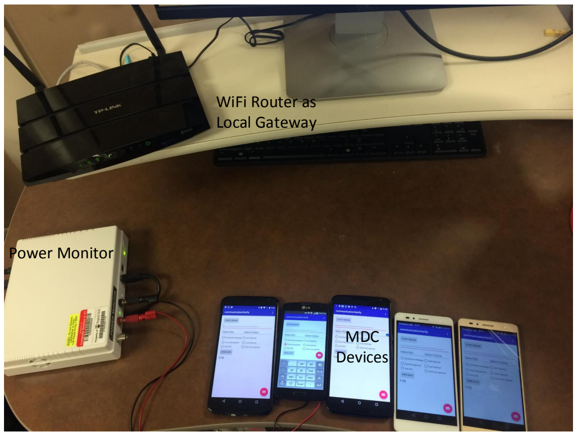
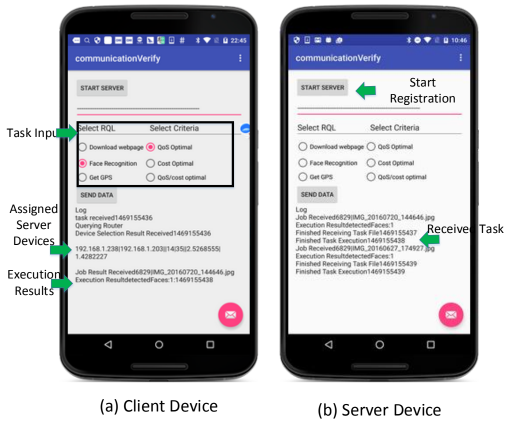


Figure 9: Hardware for the Implementation and Evaluation.

Fig.9 shows our evaluation's hardware components, which include two Nexus 6 phones, two Huawei Honor 5x, one LG Volt Phone, a Monsoon power monitor, and a TP-LINK TL-WDR3600 router. To make the WDR3600 router serve as the local gateway, we flush openWRT system image to replace the system image provided by the vendor. openWRT system is a Linux distribution for embedded devices. We further install PHP, MySQL and nginx to provide web services, and develop the corresponding PHP script files for the interfaces defined in Section V.

For evaluating MCL, we develop a distributed app, whose client and server parts run on microservice invoking devices and the MDC devices, respectively. For MDC devices, their user decides whether to start or stop sharing device capabilities via a simple button click, which sends the corresponding MCL script to the local gateway. For the microservice invoking devices, their users generate different request combinations of microservices and NFRs. We implement and evaluate three microservice packages: file download, face recognition, and get GPS. To simplify the device selection requests, we define the same NFRs for all these three microservices, namely `QoS`, `cost`, and `efficiency` (QoS/cost).

Fig.10 shows the runtime procedure of executing the microservice of face detection. The cost of performing face detection is determined by the remaining battery level: a lower battery level leads to a higher cost. The QoS of the service execution is determined by the frequency of the CPU: a higher CPU frequency leads to faster execution, and thus higher QoS. One Nexus 6 serves as the client device, and the other four devices serve as available devices. After receiving the service request, the client device first queries the connected router, and obtains the IP address of the assigned server device. It then connects to the assigned device via a socket and sends



```
Log
task received1469155436
Querying Router
Device Selection Result Received1469155436
192.168.1.238|192.168.1.203||14|35||2.5268555|
1.4282227
Job Result Received6829|IMG_20160720_144646.jpg
Execution ResultdetectedFaces:1:1469155438
```

```
Log
Job Received6829|IMG_20160720_144646.jpg
Execution ResultdetectedFaces:1
Finished Receiving Task File1469155437
Finished Task Execution1469155438
Job Received6829|IMG_20160627_174927.jpg
Execution ResultdetectedFaces:1
Finished Receiving Task File1469155439
Finished Task Execution1469155439
```

Figure 10: Execution UI.

the package's and function's names, the input parameters, and the image files to process to the server device. After execution, the results are passed back to the client device.

B. *Performance Evaluation*

a) Device Selection: For each microservice, we test different NFRs, to simulate the dissimilar requirements that can be imposed on the device selection criteria (e.g., some may want the service to be executed as fast as possible, while others may want to incur the smallest costs). When the criteria is `QoS optimal`, the Nexus 6 is selected, because it has the highest CPU frequency. When the criteria is `Cost optimal`, the LG Volt is selected, because it is connected to an external power supply.

b) Execution Time: We repeat the experimental execution 10 times, and calculate the average time taken by each procedure on the client device. We observe that, the time consumption for microservice execution device selection is low (0.15s), compared with the time cost of establishing a connection to the selected device(0.61s), and executing the microservice(1.26s). For the MDC device, the average time consumed to register its capabilities is 0.87s, because it needs to obtain the device's real-time status. Although the registration time is close to one second, this latency should not affect the perceived system performance; while the device information is being updated, the old device information can still be used simultaneously.

c) Energy Consumption: We record the energy consumption of the LG Volt device in the idle state for 30 seconds, and record the energy consumed by querying the microservice execution device/registering device capability once per second for 30 seconds. To protect the result from being distorted by the caching strategy of the Android Volley library, we add a random parameter to each request.

Our experiment shows that, the energy consumption for the client device to parse the MCL request and obtain the assigned MDC device from the WiFi router is 0.009 mAh; the energy consumed by the MDC device to register with the WiFi router is 0.023mAh. If an MDC device registers with the gateway

Number of Devices	1	10	20	50	100
Server Device Query (ms)	14	90	171	377	531
Capability Registration (ms)	18	110	192	461	563

Table I: Gateway's Average Response Time.

once per minute for one day, the overall energy consumed would be 33mAh, and this energy expense should not affect the experience of mobile users, given that the battery capacity of a typical modern smartphone is at least 2000mAh.

d) Performance of the Gateway: We use ab to benchmark the performance of the HTTP services, including registering device capability and querying for microservice execution devices, provided by the WiFi router. We run this test on a notebook that connects to the router via WiFi. We simulate 1, 10, 20, 50, 100 devices connecting to the router simultaneously, and Table I shows the average execution time. As the bulk of the processing load takes place in the WiFi router, the obtained results show high scalability even when stress testing the system with an unrealistic number of requests to the router.

C. Device Selection Procedure

We also experiment with comparing our device selection procedure with that of other state-of-the-art systems. Table II gives the description of three key competing designs with 3) being our system.

Device Discovery	Energy	Latency	Programmability
BLE Broadcast	Low	1.26s	Low
UDP Broadcast	Middle	0.38s	Low
Router as Gateway	Middle	0.2s	High

Table II: Properties of Device Selection Mechanisms.

1) BLE Broadcast Based [26]: The functionality demanding devices use the BLE broadcast to announce their requirements. When the MDC devices receive the broadcast, they connect to the broadcasting device, and transfer their device capability to it. For the broadcasting device, if multiple MDC devices can provide the required functionality, it needs to wait for all MDC devices to respond, and then select one device that best fits the NFRs, and establish a BLE connection with that device for executing functionality remotely.

2) UDP Broadcast Based [28], [29]: MDC devices are all connected to a local network. The functionality demanding device sends out a UDP broadcast, with the required functionality, the NFRs, and the IP address of the device included in the broadcast message. When an MDC device receives the broadcast and determines that it fits the requirements, it sends its information back to the broadcasting device. The broadcasting device waits for all nearby devices to respond, and then starts a socket connection with the device that best fits the NFRs.

Here we compare the performances and applicability of all the considered device selection strategies:

1. **Energy.** Table II shows the comparison of the amount of energy consumed by each strategy over time. BLE is the most energy-efficient, while the other two methods consume slightly more energy.

Execution Time	2h	4h	6h	8h
Stand By	93 %	87 %	79 %	71 %
BTLE D2D Broadcast	93%	86%	78%	70%
Node in WiFi Cluster	92%	84%	76%	68%

Table III: Remaining Battery Percentage Over Time.

ULOC	Register & Stop	Device Selection
Router-based	33	5
BLE Broadcast	86	231
UDP Broadcast	57	208

Table IV: ULOC for Each Function.

2. **Latency.** Table III shows the latency result of our experimental implementation, with an MDC comprising three devices. We conclude that 1) The latency of BLE is the highest, because all MDC devices need to connect to the resource requesting device, and pass their capacity to the device via BLE communication, which is rather slow. 2) the UDP broadcast strategy also incurs higher latency than the gateway-based ones. We further increase the number of the MDC devices to 5, and observe that the latency of both UDP and BLE broadcasts increase accordingly.

3. **Programmability.** We evaluate the programmability of these strategies, in terms of uncommented lines of code it takes to implement each functionality. When registering device capabilities, our strategy takes 33 ULOC, with the majority of the code written to obtain the device's status. The two broadcast based strategies take 57 and 86 ULOC, respectively, due to them needing to manage the D2D communication. When selecting devices, our strategy takes only 5 lines of code, with the broadcast based strategies taking over 200 ULOC.

Based on this evaluation, one can conclude that our gateway-based system architecture enables mobile apps to leverage MDCs with low latency and high energy efficiency. In addition, our architecture's device selection procedure requires fewer lines of programmer-written code as compared to the broadcast-based alternatives.

VII. Related Work

Much of recent work has focused on leveraging the resource capability (computation, sensing, and network) of the collocated mobile devices[1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [30], [11], [12], [13], [14], [15], [16], [17], [18].

Existing solutions for discovering collocated devices can be divided into two categories: 1) the device requiring the services should monitor the devices within its communication range in a p2p manner [3], [5]; 2) a device is selected as the cluster head, and handles the device discovery procedure instead [31], [9]. AllJoyn [32], a framework included in Windows 10 for enabling device-to-device communication, implements both solutions. However, the first solution suffers from poor scalability incurring high performance overhead on both the client and server devices. The second solution requires purchasing additional devices and complex setup procedure. The second solution requires writing a lot of low level codes to manage the communication among mobile devices. Another recent closely related work also implements a system architecture that uses generally available, off-the-shelf WiFi routers as a gateway for device discovery [33]. However, that solution fails to select device according to requirements on execution features.

VIII. Conclusion

In this paper, we have presented a novel system architecture for mobile device clouds (MDCs). The architecture adapts the microservice pattern to MDC environments, and offers an intuitive programming model for MDC applications. Developers interact with the architecture via a high-level programming abstraction in the form of a domain-specific language. The language concisely expresses the start and stop of device capability sharing as well as the selection of the most suitable devices to execute a given functionality. The results of evaluating our architecture's reference implementation show how its efficiency and programmability can make it a viable solution for leveraging MDCs.

Acknowledgment

This research is supported by the National Science Foundation through the Grants CCF-1649583 and 1717065.

References

[1] A. Mtibaa, A. Fahim, K. A. Harras, and M. H. Ammar, "Towards resource sharing in mobile device clouds: Power balancing across mobile devices," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 51–56.

[2] E. E. Marinelli, "Hyrax: cloud computing on mobile devices using mapreduce," DTIC Document, Tech. Rep., 2009.

[3] G. Huerta-Canepa and D. Lee, "A virtual cloud computing provider for mobile devices," in *MCS'10*. ACM, p. 6.

[4] E. Miluzzo, R. Cáceres, and Y.-F. Chen, "Vision: mclouds-computing on clouds of mobile devices," in *MCS'12*. ACM, 2012, pp. 9–14.

[5] C. Shi, V. Lakafosis, M. H. Ammar, and E. W. Zegura, "Serendipity: enabling remote computing among intermittently connected mobile devices," in *MobiHoc'12*. ACM, 2012, pp. 145–154.

[6] A. Fahim, A. Mtibaa, and K. A. Harras, "Making the case for computational offloading in mobile device clouds," in *MobiCom'13*. ACM, 2013, pp. 203–205.

[7] U. Drolia, R. Martins, J. Tan, A. Chheda, M. Sanghavi, R. Gandhi, and P. Narasimhan, "The case for mobile edge-clouds," in *UIC/ATC'13*. IEEE, 2013, pp. 209–215.

[8] K. Bhardwaj, S. Sreepathy, A. Gavrilovska, and K. Schwan, "Ecc: Edge cloud composites," in *MobileCloud'14*. IEEE, 2014, pp. 38–47.

[9] R. Loomba, R. de Frein, and B. Jennings, "Selecting energy efficient cluster-head trajectories for collaborative mobile sensing," in *GLOBECOM'15*. IEEE, 2015, pp. 1–7.

[10] X. Chen, "Decentralized computation offloading game for mobile cloud computing," *IEEE TPDS*, vol. 26, no. 4, pp. 974–983, 2015.

[11] E. Toledano, D. Sawada, A. Lippman, H. Holtzman, and F. Casalegno, "Cocam: A collaborative content sharing framework based on opportunistic p2p networking," in *CCNC'13*. IEEE, 2013, pp. 158–163.

[12] S. Sur, T. Wei, and X. Zhang, "Autodirective audio capturing through a synchronized smartphone array," in *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. ACM, 2014, pp. 28–41.

[13] A. Amiri Sani, K. Boos, M. H. Yun, and L. Zhong, "Rio: a system solution for sharing i/o between mobile systems," in *MobiSys'14*. ACM, 2014, pp. 259–272.

[14] H. Liang, H. S. Kim, H.-P. Tan, and W.-L. Yeow, "Where am i? characterizing and improving the localization performance of off-the-shelf mobile devices through cooperation," in *NOMS'16*. IEEE, 2016, pp. 375–382.

[15] J.-W. Yoo and K. H. Park, "A cooperative clustering protocol for energy saving of mobile devices with wlan and bluetooth interfaces," *TMC*, vol. 10, no. 4, pp. 491–504, 2011.

[16] X. Bao, Y. Lin, U. Lee, I. Rimac, and R. R. Choudhury, "Dataspotting: Exploiting naturally clustered mobile devices to offload cellular traffic," in *INFOCOM'13*. IEEE, 2013, pp. 420–424.

[17] B. Jones, K. Dillman, R. Tang, A. Tang, E. Sharlin, L. Oehlberg, C. Neustaedter, and S. Bateman, "Elevating communication, collaboration, and shared experiences in mobile video through drones," in *DIS'16*. ACM, 2016, pp. 1123–1135.

[18] E. Koukoumidis, D. Lymberopoulos, K. Strauss, J. Liu, and D. Burger, "Pocket cloudlets," in *SIGPLAN*, vol. 46, no. 3. ACM, 2011.

[19] A. Salem and T. Nadeem, "Colphone: A smartphone is just a piece of the puzzle," in *UbiComp'14*. ACM, 2014, pp. 263–266.

[20] A. Lucero, J. Clawson, K. Lyons, J. E. Fischer, D. Ashbrook, and S. Robinson, "Mobile collocated interactions: From smartphones to wearables," in *CHI'15*. ACM, 2015, pp. 2437–2440.

[21] H. J. La and S. D. Kim, "A conceptual framework for provisioning context-aware mobile cloud services," in *IEEE CLOUD*. IEEE, 2010, pp. 466–473.

[22] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, and B. Koldehofe, "Mobile fog: A programming model for large-scale applications on the internet of things," in *Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing*. ACM, 2013, pp. 15–20.

[23] K. Habak, M. Ammar, K. A. Harras, and E. Zegura, "Femto clouds: Leveraging mobile devices to provide cloud service at the edge," in *CloudCom'15*. IEEE, 2015, pp. 9–16.

[24] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *Computing Colombian Conference (10CCC), 2015 10th*. IEEE, 2015, pp. 583–590.

[25] N. Viennot, M. Lécuyer, J. Bell, R. Geambasu, and J. Nieh, "Synapse: a microservices architecture for heterogeneous-database web applications," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 21.

[26] T. Le Vinh, S. Bouzefrane, J.-M. Farinone, A. Attar, and B. P. Kennedy, "Middleware to integrate mobile devices, sensors and cloud computing," *Procedia Computer Science*, vol. 52, pp. 234–243, 2015.

[27] S. Zheng, L. Minh, K. Young-Woo, and T. Eli, "Extemporaneous micro-mobile service execution without code sharing," in *HotPOST'17*.

[28] G. Jie, C. Bo, Z. Shuai, and C. Junliang, "Cross-platform android/ios-based smart switch control middleware in a digital home," *Mobile Information Systems*, vol. 2015, 2015.

[29] J. Lee, H. Lee, Y. C. Lee, H. Han, and S. Kang, "Platform support for mobile edge computing," in *IEEE CLOUD'17*. IEEE, 2017, pp. 624–631.

[30] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," *IEEE/ACM TON*, 2015.

[31] N. Fernando, S. W. Loke, and W. Rahayu, "Dynamic mobile cloud computing: Ad hoc and opportunistic job sharing," in *UCC'11*. IEEE, 2011, pp. 281–286.

[32] Y. Wang, L. Wei, Q. Jin, and J. Ma, "Alljoyn based direct proximity service development: Overview and prototype," in *Computational Science and Engineering (CSE), 2014 IEEE 17th International Conference on*. IEEE, 2014, pp. 634–641.

[33] J. Gedeon, C. Meurisch, D. Bhat, M. Stein, L. Wang, and M. Mühlhäuser, "Router-based brokering for surrogate discovery in edge computing," in *HotPOST'17*, Jun. 2017, pp. 1–6.