2017 IEEE 37th International Conference on Distributed Computing Systems Workshops

Extemporaneous Micro-Mobile Service Execution Without Code Sharing

Zheng Song*, Minh Le†, Young-Woo Kwon†, and Eli Tilevich*

*Software Innovations Lab., Virginia Tech Email: {sonyyt,tilevich}@cs.vt.edu
†Dept. of Computer Science, Utah State University
Email: minh.le@aggiemail.usu.edu, young.kwon@usu.edu

***Abstract*—In mobile edge computing, a mobile or IoT device requests a nearby device to execute some functionality and return back the results. However, the executable code must either be pre-installed on the nearby device or be transferred from the requester device, reducing the utility or safety of device-to-device computing, respectively. To address this problem, we present a micro-service middleware that executes services on nearby mobile devices, with a trusted middleman distributing executable code. Our solution comprises (1) a trusted store of vetted mobile services, self-contained executable modules, downloaded to devices and invoked at runtime; and (2) a middleware system that matches service requirements to available devices to orchestrate the device-to-device communication. Our experiments show that our solution (1) enables executing mobile services on nearby devices, without requiring a device to receive executable code from an untrusted party; (2) supports mobile edge computing in practical settings, increasing performance and decreasing energy consumption; (3) reduces the mobile development workload by reusing services.**

I. Introduction

Mobile and IoT devices often need to make use of external computational resources. Applications for these devices may rely on functionality that cannot be implemented using the resources of a single device. One option for enhancing the execution capacities of a resource-limited device is to use cloud-based services [10]. The explosion of sensory data has given rise to *edge computing*, processing the collected sensory data locally at the edge of the network. In *mobile edge computing*, devices accomplish many of the computational tasks by networking with each other directly. This arrangement also prevents the bandwidth bottleneck of the devices simultaneously communicating with a remote cloud-based server.

However, before a device can execute any functionality, it needs the executable files. Currently, there are two options for obtaining these files: (1) install them in advance, (2) transfer them at runtime from the requesting device. The first option substantially limits the utility of mobile edge computing. Oftentimes, mobile execution environments are highly dynamic, and it is not even known a priori which devices will be operated in the vicinity. The second option compromises security. In terms of potential security vulnerabilities, it would be too risky to accept executable code from an untrusted party. In summary, the realities of mobile execution impose several technical obstacles on mobile and IoT devices sharing resources with nearby devices:

- *Trust/privacy*—the users of nearby devices may be willing to perform a computation on behalf of unfamiliar clients, but they are unlikely to trust these clients enough to accept code from them for execution;
- *Dynamicity*—at the time when the application needs to make use of an external resource, it is impossible to predict what kind of mobile devices will happen to be nearby and what resources they will have at their disposal;
- *Heterogeneity*—it is impossible to predict which mobile platforms the available nearby devices will run.

This paper describes the Mobile Service Market (MSM), a trusted remote store of vetted mobile services, each of which constitutes a self-contained executable module to be downloaded to mobile devices and invoked at runtime. The MSM can solve the problem as follows. When developing an application, a mobile developer realizes that the device on which the application will run may not possess the required functionality. She then browses a mobile services market (MSM) to find a service with the required functionality. The developer specifies the requirements for a device that can be selected at runtime to execute the service. The requirements are configured by the programmers to indicate the minimal properties of the device that can be selected to execute the service. This paper makes the following contributions:

- *Micro-mobile services*—a service-oriented solution to the resource scarcity problem of mobile devices.
- *Mobile Service Market*—a novel system architecture for hosting and deploying device-to-device mobile services on demand, thus alleviating the issues pertaining to a lack of trust/privacy.
- *Empirical Evaluation*—We rigorously evaluate the effectiveness and performance/energy efficiency of our reference implementation on a series of micro and macro benchmarks and applications.

The rest of this paper is organized as follows. Section II describes device-to-device mobile services in detail. Section III presents our evaluations results. Section IV introduces the related works, and Section V presents future work directions and concludes the paper.

2332-5666/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDCSW.2017.70

II. Micro-Mobile Services

Micro-mobile services[1] extends the notion of the Service Oriented Architecture (SOA) for the needs of ad-hoc mobile execution on nearby devices. Unlike a traditional service that is hosted at a location identified by a fixed domain name (e.g., an IP address), mobile services reside at a mobile service market and deployed on devices for execution on demand at runtime. Since the platform on which a mobile service will be executed is unknown until the runtime, service developers are expected to provide several equivalent versions for each service to support execution on all major platforms. Since mobile services are expected to be executed on devices with limited resources, service developers have to design their solution with resource scarcity in mind. The acceptance criteria for hosting a service in a Mobile Service Market must be necessarily more stringent than those for accepting mobile applications to application markets.

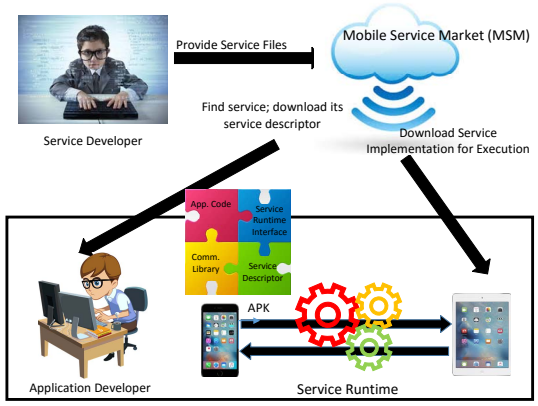

Figure 1: Proposed MSM Architecture.

As shown in Fig. 1, our solution contains three novel components: (1) The Mobile Service Market (MSM): an online service market for hosting and deploying mobile services; (2) Service Middleware that matches services with suitable devices at runtime as well as manages the communication across heterogeneous devices; (3) Programming model that provides a convenient interface for mobile application developers to choose the needed services and configure their requirements. We next describe the novel parts of our contribution in turn.

A. Service Market

Figure 1 shows the main components of the MSM architecture that codifies interactions across three different roles: service developer, application developer, and mobile user.

The mobile service developer identifies those pieces of application functionality that can be represented as mobile services. A service submitted to a service market must adhere to a format detailed below.

The application developer incorporates mobile services into their applications. To that end, they need to browse through the catalogs of mobile services of the MSMs that their users are likely to trust. They select the services that solve the resource scarcity problem at hand, and are able to change the constraints of the invoked services for the specific needs of their applications.

The mobile user will need to configure their device to be willing to accept the execution of services, dynamically downloadable from a given MSM. The middleware will manage the service's lifecycle, such as obtaining the latest version of the service execution package for further invocations.

In essence, the MSM combines the features of application markets and mobile service repositories. Following the application market model makes it possible for users to rely on the reputation of a given market to have enough trust to allow the automatic installation and execution of such mobile services. At the same time, service developers would have to comply with the service market requirements, which likely would have to be more stringent than those of application markets.

1) Service Representation: Our design of MSM defines a typical mobile service as a collection of three elements:

Service Description: uniquely identifies a service as a combination of service name, version, usage scope, and parameters.

Service Execution Package (SEP): is a self-contained executable package that can be downloaded from the service market and executed on a mobile device. A service can be designed for one particular platform, several platforms, or all platforms by means of JavaScript execution. Platform availability is one of the selection criteria that mobile developers need to consider when deciding to use a mobile service in their applications.

Constraints: are requirements on the device that can be selected by the runtime to execute a service. Constraints are defined by service developers, with some of their parameters configured by application developers for the needs of a given application. Currently, our reference implementation makes use of the following constraints:

- Sensor availability: `REQ` or `NREQ`. For example, a service may require a GPS sensor for execution.
- Battery threshold: `N(%)` – device does not respond to a service execution request if the remaining battery level is lower than $N\%$.
- Expected QoS Level: `(N)` – the detailed definition of expected QoS level (EQS) will be given in the next subsection. Generally speaking, EQS is a metrics of a mobile device's resource status.
- Network availability: `HIGH`, `LOW`, or `N.A.`
- Number of required devices: `N`, the number of required mobile devices to execute a service (e.g, $N > 1$: collaborative execution)

B. Service Execution Model (Middleware)

The middleware system provides a communication infrastructure for mobile devices, coordinating the execution of services between clients and servers. In this section, we explain each component of the middleware system in turn.

[1]For brevity, in the rest of the presentation we shall use the terms *micro-mobile services* and *mobile services* interchangeably.

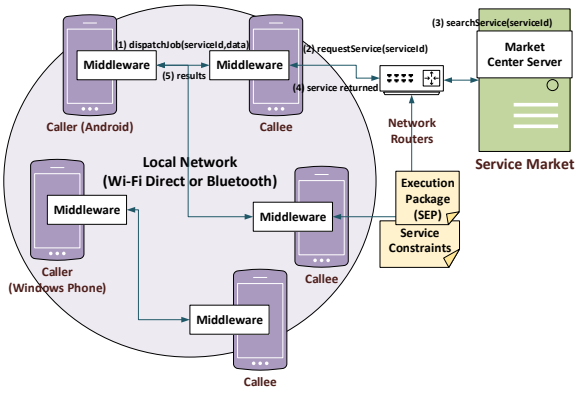

Figure 2: Service Market overview

1) Discovering Available Nearby Devices: To discover available devices, a client device first sends out a peer discovery message to nearby devices, and then each device replies with its availability that represents resource capacities. The following JSON format shows a response message to the peer discovery message.

```
JSON : DeviceInfo {
  "availability": Boolean, "gps": Boolean,
  "network": ["low"|"high"|"off"], "EQS": Double}
```

where `availability` indicates whether the device is ready for any execution; `gps` shows the availability of GPS; `network` shows the network state, which is either `high`, `low` or `off`; and `EQS` is used for expressing the level of service execution capacity, which will determine service quality.

2) Selecting Available Devices: The middleware system's ability to select the most suitable devices is crucial for ensuring high performance and low energy consumption. However, in opportunistic networks comprising arbitrary mobile devices, the peer selection problem has been a deep, fundamental research problem for the last decade [5]. To address this problem, we introduce a quality- and constraints-based peer selection mechanism that works as follows. First, after collecting `EQS` values from nearby devices, the client updates these values with the latest latency information. Then, if a service requires only one device, the middleware system selects the device that has the largest `EQS` value and meets the other selection criteria. Otherwise, it selects a number of suitable devices following the descending order of `EQS` values. To limit the number of selected devices to avoid possible performance or energy efficiency overheads, we use the following strategy: (1) first filter out low-end mobile devices that have lower resource capacities than the client (i.e., 20% lower `EQS` than the client's `EQS`); (2) choose the number of servers based on following straightforward equation, which is obtained heuristically.

Computing EQS: Once the client device has collected all the response messages from nearby devices, the most favorable devices for the given service execution are selected in accordance with the `EQS` value defined in the previous section. The `EQS` value is computed using the resource usage information including CPU, memory, battery, and network. An initial `EQS` value is calculated using the device-specific information such as CPU, memory and battery, and then updated with an end-to-end latency on the client device's side as follows:

$$EQS = \quad (CPU_{Avail} + Mem_{Avail} + Batt_{Avail}) \times Net_{RTT} \tag{1}$$

where CPU_{Avail}, Mem_{Avail}, and $Batt_{Avail}$ are the ratios of available resources, respectively. Net_{RTT} is a round-trip time it takes to send a peer discovery message and receive a corresponding response between a client and nearby devices. Since the round-trip time is only measured on the client side, each nearby device sends an initial `EQS` value along with a response message. Then, the client re-calculates the `EQS` with the actual round-trip time. Each resource usage information can be collected and quantified using standard system APIs.

3) Partitioning Data: Once the list of devices are identified, the middleware system divides a job into N equal tasks. To make the data partitioning problem simple, we adopted the Hadoop's idea that splits data into multiple chunks of the same size, so that the devices that have more hardware capacities will finish their tasks earlier than others and accept more tasks.

4) Service Execution and Fault Handling: The service execution procedure contains the following steps: (1)the client send the service execution request to the selected server devices; (2) the server devicess download the service execution package from MSM, and execute the SEP; (3) the server devices send the service execution results back to the client.

Due to the volatile nature of mobile networks, failures are a constant presence of mobile execution. As a failure handling strategy for mobile service execution, the Service Middleware on the client listens to the network-related updates (e.g., network join/leave events from `BroadcastReceiver` class). When there are any failures reported, the middleware, if possible, will attempt to by resume the service execution locally at the client for any reported failures. A simple checksum mechanism is used to verify the integrity of the execution results.

C. Development Support

1) For Mobile Application Developers: To become a pragmatic solution to the resource scarcity problem of mobile devices, device-to-device mobile services must provide a convenient programming model to the mobile developers for them to invoke mobile services in their applications. In our reference implementation, we experimented with integrating the notion of mobile services with a modern Integrated Development Environment (To support custom tools, modern IDEs offer an extensibility mechanism realized as plug-ins). The provided IDE plug-in provides three basic functionalities: searching for mobile services, generating sample code for invoking mobile services, and specifying service constraints.

2) For Mobile Service Developers: To support the heterogeneity across different platforms and devices, mobile services are written in JavaScript, and are executed by a JavaScript engine. JavaScript service implementations lack access to system resources (e.g., local files, networks, sensors etc.). To make it possible to access local system resources from JavaScript code, we provide Java interfaces by means of the Java System Resource interface (JSRI). Next, we describe how we implemented two different JavaScript engines for the Android and Windows platforms.

a) Native Mobile Services: To implement the native version of *Service Definition*, a programmer must implement a service execution package using the provided APIs. In particular, the programmer needs to explicitly provide a service execution class and data processing class for the service (e.g., how to serialize/deserialize data, split or merge partial data etc.). Then, all the classes are compiled to the DEX format and put into a JAR file, which is added into *Service Execution Package* with the *service descriptor* and *constraints* files and uploaded to Service Market. Once a worker receives a *Service Execution Package*, a JAR file is extracted from the package and loaded by searching a service descriptor in the package.

b) JavaScript-based Mobile Services: To support the heterogeneity across different platforms and devices, the *Service Definition* must be written in JavaScript, which are executable on any platforms with compatible JavaScript engines. In JavaScript-based services, a JavaScript file is deployed as a service package and then executed by a JavaScript engine. To support heterogeneous mobile devices, we have developed execution engines for Android, Windows, and Linux platforms.

c) Mobile Services for Android and Windows Platforms: To execute a service execution package on the Android platform, the runtime system instantiates a local web server using NanoHttpd [2], so that any Android device can load the extracted micro-mobile service written in JavaScript through a built-in JavaScript engine (i.e., WebView). For the Windows platform, we implemented our JavaScript engine using default WebBrowser and local web server[3].

III. Evaluation

We have evaluated the effectiveness of our approach through a micro-benchmark and four realistic case studies. Specifically, to evaluate the performance and energy efficiency of each device in its network environment, we implemented three non-trivial test cases to ascertain how effective the system would be in enabling mobile developers to expend a moderate programming effort to extend the functional capabilities of their applications.

- **Image Processing Service:** In this service, an image is split into several parts, and each device then blurs one image part and sends the result back to the client.
- **Internet Sharing Service:** A device without Internet access can access webpages with support from the surrounding internet-enabled devices.

[2]http://nanohttpd.org/

[3]https://github.com/ideaconnect/wp8-simple-web-server

- **GPS Sharing Service:** GPS location can be obtained from a nearby device.

Table I shows the testing devices used in the experiments.

Table I: Specifications of the testing devices

Devices	CPU	RAM	Battery	OS
LG Opt. GK	1.7GHz	2GB	3100mAh	4.4.2
LG G Stylo	1.2GHz	1GB	3000mAh	6.0
LG Tribute	1.2GHz	1GB	2100mAh	4.4
LG G4	1.44GHz	3GB	3000mAh	5.1
Galaxy S3	1.4GHz	1GB	2100mAh	4.4
Asus Zenfone 2	1.7GHz	3GB	2400mAh	5.1
LG G Pad	1.7GHz	2GB	4600mAh	4.2.2
Lumia 550	1.1GHz	1GB	2100mAh	W. 10

A. Micro-Benchmarks

We measured the service initiation time includes (1) time for requesting a service and receiving a response message, (2) time for parsing the service response message and description, time for downloading the service package (SEP), which is either a JavaScript or a native version, and (4) time for installing the downloaded service package. We tested the image processing service from two different places Asia (Vietnam) and US on the same specification devices to see how the geographic distance affects the service retrieval time. Table II shows the service downloading time for both scenarios. The file size of native version and JavaScript version is 2.07kb and 1.29kb, respectively. The JavaScript version does not have any installation time because it is interpreted at runtime while the native version includes the classloading time.

Table II: Average service initiation time in different locations: (1) Asia—U.S. and (2) within LAN

JS (1)	Native (1)	JS (2)	Native (2)
1153ms	1775ms	809ms	896ms

B. Test Case Results

1) Image Blurring Service: To compare the performance and energy consumption between a native- and a JavaScript-based version, we implemented two different versions of the image blurring service in JavaScript and Java, and then we selected a low-end device (Galaxy S3) as the client, while other mid- and high-end devices (LG G4, LG GK, Asus and Tribute) are nearby devices. As depicted in Figure 3, the JavaScript-based service could improve the performance by the maximum 38% when executing the mobile service on multiple devices, while the native version improved the performance by 42—68%. The bottom graph of Figure 3 also shows the similar energy consumption pattern between the two versions. To process the binary data, the JavaScript version requires additional processing to convert the binary data of an image into plain strings which is one of the main reason the native version outperforms the JavaScript version.

2) Internet Sharing Service: We designed the Internet sharing service to distribute requests including *URL*, *n* – number of devices in the cluster and *index* – index of a device. Each device (include the client) downloads the HTML text contents

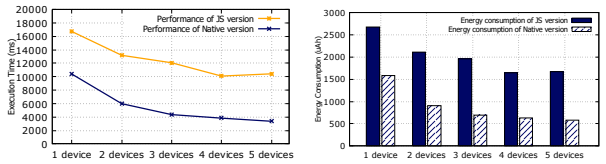

Figure 3: The performance and energy consumption comparison between the native and JavaScript version of the Image-processing service

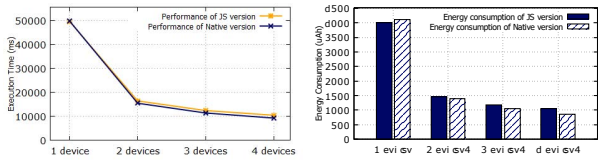

Figure 4: The performance and energy consumption comparison between the native and JavaScript version of the Internet-sharing service

of a web-page from the *URL*, collects its resource URLs (images, audio, videos etc.) and downloads a batch of URLs according to its *index* position of *n* devices.

Figure 4 shows the experimental result when downloading HTML pages through nearby devices using the native and JavaScript versions. Although the native version shows the better efficiency in both experiments, the differences are relatively small because the web-page downloading and data transmission have been processed through the JSR interface, whereas JavaScript only contributed as the demand and constraint solving carrier.

3) GPS Sharing Service: In this experiment, we performed three GPS test cases on the native version: two for remote GPS requests and one for local GPS (invoke by itself). For each case, we used two devices with the same specifications and fixed them as the client and server to minimize the overloads throughout the system. Figure 5 shows our experimental results that GPS Sharing fails to improve performance. Indeed, with the client having the same resource specs, accessing a remote GPS sensor and transmitting data back and forth takes longer than accessing the local GPS sensor. However, when the local GPS sensor is unavailable, accessing the GPS sensor of a nearby device provides a viable alternative.

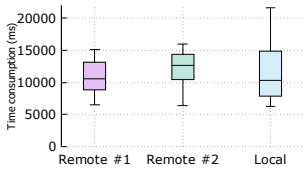

Figure 5: The performance comparison of the GPS sharing.

C. Programming Effort

Here we estimated the programming effort required to use our mobile services infrastructure, which comprises 5K ULOC of Java and JavaScript to implement the middleware system and 1K ULOC for our MSM implementation. Since the same developer was in charge of creating services, both their cross-platform (in JavaScript) and native (in Java and C# for Android and Windows Phone, respectively) versions, we report the combined effort it took to develop the services and the client code that invokes them. Specifically, we calculated the total number of uncommented lines of code and an approximate number of days it took to develop, troubleshoot, and optimize each case study. The co-author who developed our case studies is an experienced Software Engineer with several years of industry experience and prior exposure to mobile application development both in academic and real-world settings. The results appear in Table III.

Table III: Total lines of code (LOC) and development time for the case studies.

Service	ULOC	Time
Image Blurring: Java	830	2 days
Image Blurring: C#	920	5 days
Image Blurring: JavaScript	725	3 days
Internet Sharing: Java	585	2 days
Internet Sharing: JavaScript	680	3 days
GPS Sharing: Java	545	2 days

Mobile application developers can use mobile services either during new development or as an optimization. If the source code is not available, one can take advantage of byte-code engineering and integrate our system with frameworks such as RetroSkeleton [7] or Rio [4], whose interception mechanisms capture requests for system resources, making it possible to specify custom responses without requiring access to the application's source code. Using these frameworks, one can, for example, redirect GPS location requests to nearby devices without modifying the source code.

D. Discussion

The micro-mobile services make it possible to design mobile applications that take advantage of complimenting strengths of nearby devices from the resource allocation perspective. Developers can orchestrate the execution of an application's functionalities on the devices that have the most suitable and abundant resources for executing them.

This software architecture removes the need to accept code for execution from nearby devices. Downloading vetted service code from trusted markets increases the trustworthiness of the distributed execution model. At the same time, mobile service developers would have to comply with the service market requirements, which would have to be more stringent than those of application markets.

In general, the MSM architecture can potentially lead to a new type of mobile developers, domain experts who excel at backend-related tasks and who may not be particularly interested in developing end-user applications.

Once a service is downloaded, the Internet connection is no longer required, making it possible to execute device-to-device services in environments with limited or intermittent wide-area networks. In fact, this architecture can increase the utilization of nearby mobile devices in such execution environments.

Finally, by addressing resource scarcity, this architecture can become a viable solution for orchestrating the execution of IoT setups, in which each participating device is likely to posses specialized functionality (e.g., sensor, media capture, etc.) while lacking general hardware or software resources.

IV. Related Work

This work is related to other complementary efforts that optimize mobile applications' performance and energy efficiency via remote execution, including special software libraries, code migration, and computation offloading.

Alljoyn [3] is an open source framework that hides the complexity of network communications for application programmers. By providing interoperability between multiple platforms without any transport layers, Alljoyn makes the integration and initiation of network communication easy and straightforward. Before the Wi-Fi Direct technology, many efforts have focused on optimizing peer-to-peer network based on existing short-range/wireless communication technologies available on mobile devices including Bluetooth, Wireless IEEE802.11 and cellular communication link [9], [11].

In addition, our approach shares objectives and techniques with one that migrates different code bases into a system. In particular, code migration can be used to update existing, legacy systems [8]. Similar to our approach, code migration mechanisms are mainly used to run code in different memory spaces (e.g., running C++ code on multi-core systems [6], running JavaScript code on a server [13], object-level migration for distributed systems [14], thread-level migration through middleware [12]). These code migration approaches have influenced the design of offloading mechanisms in the mobile computing area.

One can compare our work with cross-platform hybrid frameworks, such as Apache Cordova (formerly PhoneGap) [1] and React Native [2]. Cordova uses standard Web technologies for cross-platform mobile development, while React Native supports iOS native development with JavaScript. Similarly to our approach, Cordova executes applications in mobile web browsers on multiple platforms, while React Native introduces a bridge library (i.e., Java System Resource interface (JSRI) in our approach) to access local system resources from JavaScript code. Despite also relying on web technologies for cross-platform execution, our approach differs from Cordova and React Native. Our approach's target is heterogeneous service execution rather than application execution.

V. Conclusion

In this paper, we presented a micro-mobile service architecture and reference implementation that execute services on nearby mobile devices through device-to-device channels. Our solution centers around a trusted store of vetted mobile services, downloaded to a mobile device and invoked at runtime, as well as a middleware system for selecting suitable devices to execute services and managing the invocation of services across devices. Our experimental evaluation shows how executing mobile services on nearby devices can improve performance and reduce energy consumption.

Acknowledgment

This research is supported by the National Science Foundation through the Grant CCF-1649583.

References

[1] Apache Cordova. https://cordova.apache.org/.

[2] React Native. http://www.reactnative.com/.

[3] AllSeen Alliance. Alljoyn framework. https://allseenalliance.org/framework.

[4] A. Amiri Sani, K. Boos, M. H. Yun, and L. Zhong. Rio: a system solution for sharing i/o between mobile systems. In *Proceedings of the 12^{th} Annual International Conference on Mobile Systems, Applications, and Services*, 2014.

[5] D. R. Choffnes and F. E. Bustamante. Taming the torrent: a practical approach to reducing cross-isp traffic in peer-to-peer systems. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 363–374. ACM, 2008.

[6] P. Cooper, U. Dolinsky, A. F. Donaldson, A. Richards, C. Riley, and G. Russell. Offload – automating code migration to heterogeneous multicore systems. In *Proceedings of the 5^{th} International Conference on High Performance Embedded Architectures and Compilers*, 2010.

[7] B. Davis and H. Chen. RetroSkeleton: retrofitting android apps. In *Proceedings of the 11^{th} Annual International Conference on Mobile Systems, Applications, and Services*, 2013.

[8] W. Emmerich, C. Mascolo, and A. Finkelstein. Implementing incremental code migration with XML. In *Proceedings of the 22^{nd} International Conference on Software Engineering*, 2000.

[9] H. C. Frank H. P. Fitzek. *Mobile Peer-to-peer (P2P): A Tutorial Guide*. Wiley, 2009.

[10] Y.-W. Kwon and E. Tilevich. Energy-efficient and fault-tolerant distributed mobile execution. In *Proceedings of the 32^{nd} International Conference on Distributed Computing Systems (ICDCS '12)*, June 2012.

[11] R. M. P. Bellavista, A. Corradi and C. Stefanelli. Context-aware middleware for resource management in the wireless internet. In *IEEE Transactions on Software Engineering*, pages 1086–1099. IEEE, 2003.

[12] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten. Portable support for transparent thread migration in Java. In *Proceedings of the 2^{nd} International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents*, 2000.

[13] T.-L. Tseng, S.-H. Hung, and C.-H. Tu. Migratom.Js: A JavaScript migrajcrtion framework for distributed web computing and mobile devices. In *Proceedings of the 30^{th} Annual ACM Symposium on Applied Computing*, 2015.

[14] M. Yoshida and K. Sakamoto. Code migration control in large scale loosely coupled distributed systems. In *Proceedings of the 4th International Conference on Mobile Technology, Applications, and Systems and the 1st International Symposium on Computer Human Interaction in Mobile Technology*, Mobility '07, 2007.