POEM: On Establishing A Personal On-demand Execution Environment for Mobile Cloud Applications

Huijun Wu, Dijiang Huang
School of Computing, Informatics, and Decision Systems Engineering
Arizona State University
{*Huijun.Wu, Dijiang.Huang*}*@asu.edu*

Min Chen
School of Computer Science and Technology
Huazhong University of Science and Technology
minchen@ieee.org

***Abstract*—A distributed mobile cloud service model called "POEM" is presented to manage the mobile cloud resource and compose mobile cloud applications. POEM provides the following salient features: (a) it considers resource management not only between mobile devices and clouds, but also among mobile devices; (b) it utilizes the entire mobile cloud system as the mobile application running platform, and as a result, the mobile cloud application development is significantly simplified and enriched; and (c) it addresses the interoperability issues among mobile devices and cloud resource providers to allow mobile cloud applications running cross various cloud virtual machines and mobile devices. The proposed POEM solution is demonstrated by using OSGi and XMPP techniques. Our performance evaluations demonstrate that POEM provides a true elastic application running environment for mobile cloud computing.**

***Keywords*-mobile cloud computing; offloading; service oriented architecture; OSGi; XMPP**

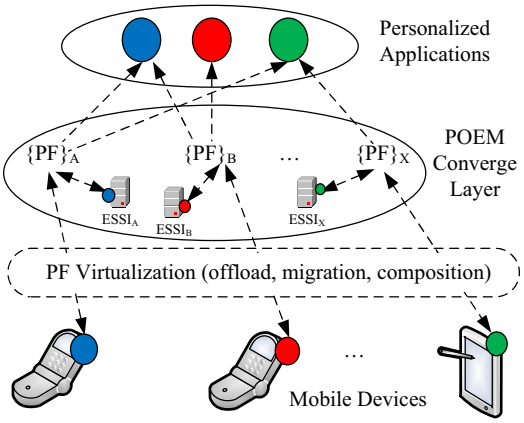


Figure 1. Overview of POEM.

I. Introduction

An ideal mobile cloud application running system should enable mobile devices to easily discover and compose cloud resources for its applications. From mobile resource providers' perspectives, they may not even know what applications are using their resources and who may call their provisioned functions beforehand. In this way, the mobile application design should not be application-oriented; instead, it should be functionality-oriented (or service-oriented). For example, the video function of a mobile device should provide general calling interfaces that can be called by multiple local or remote functions in the runtime. To achieve this feature, we can consider these Provisioning Functions (PFs) as the fundamental application components in the mobile cloud, which can be composed by mobile cloud service requesters in realtime. As a result, mobile cloud can significantly reduce the mobile application development overhead and greatly improve the agility and flexibility to build a personalized mobile cloud computing system that can be customized for each mobile user.

There are several challenges in current mobile cloud application scenarios. The first challenge is that knowing the status of mobile devices, e.g., online/offline and runtime information (such as battery, computing power, connectivity, etc.), is difficult due to the mobility of mobile users. The second challenge is that knowing the available PFs on each mobile device is not a trivial task. Currently, there is no such a common framework allowing mobile devices for exchanging the available PFs information and running such a system in a distributed environment. The third challenge is to compose PFs crossing various hardware and software platforms, which demands a universal programming and application running environment with little compatibility issues.

To address these challenges, we present a new mobile cloud application running system, which is called POEM (Personal On-demand execution Environment for Mobilecloud computing), as shown in Figure 1. POEM treats each mobile device as a PF provider. In addition, POEM is designed based on the mobile cloud framework, where a dedicated Virtual Machine (VM) is assigned to each mobile device providing computing and storage support. Moreover, PFs can be offloaded/migrated from a mobile device to its assigned VM. Thus, the VM can not only run mobile devices' PFs (i.e., as shadows), but also can it run extended PFs that mobile devices may not have the capacity to execute. Thus, we also call the VM in the

978-1-4673-7284-8/15 $31.00 © 2015 IEEE
DOI 10.1109/MS.2015.16

POEM framework as ESSI (Extended Semi-Shadow Image). Collectively, the PFs provided by a mobile device X and its corresponding $ESSI_X$ is denoted as $\{PF\}_X$. POEM regards mobile devices and their dedicated ESSIs both as PF providers. As a result, the mobile user's applications can be composed by PFs from local PFs (may be offloaded/migrated to its dedicated ESSI) and/or remote PFs (may run on remote mobile devices or their dedicated ESSIs).

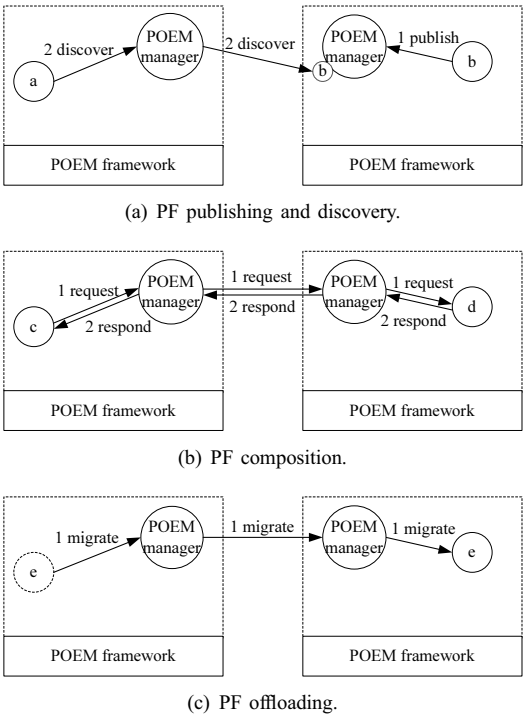
To demonstrate the proposed POEM solutions, we implemented a pilot POEM system based on OSGi [1] and XMPP [2] techniques. In summary, the contributions of the presented research is highlighted as follows:

- *Social mobile cloud computing:* POEM solution enables mobile cloud application to utilize social network power, i.e., in addition to the discovered PFs through the mobile cloud system, mobile user can establish mobile cloud applications through their trusted social connections. In this way, POEM applications not only can use the resource in cloud by offloading resource intensive components but also can use services provided from their social connections.
- *Versatile and personalized application offloading, migration, and composition:* POEM maintains available mobile cloud resource and allows users choosing a mobile cloud application by using different approaches (offloading, migration, and composition) based on the available system resources and their personalized application requirements.

The paper is organized as follow. Section II introduces related work. Section III describes systems and models. Sections IV and V discuss POEM design and implementation, respectively. Section VI presents evaluation results. Finally, Section VII concludes the paper.

II. Related Work

Most of the previous research focuses either on mobile tasks partition and composition or on offloading techniques. μCloud [3] describes a framework for mobile cloud application composition from heterogeneous software components. μCloud is a static mobile cloud application model, which requires a lot of work for programmers to partition application and decides which partition runs on which part of the cloud. eXCloud [4] focuses on offloading and migration: it migrates Java Virtual Machine (JVM) runtime to cloud. However it migrates only the top portion of runtime stack, rather than the whole virtual machine, to cloud using Stack On Demand (SOD) [5] migration technique. CloneCloud [6] maintains a clone of mobile device in the cloud. CloneCloud can deal with dynamic offloading, however it requires synchronization between mobile device and cloud, which is not always satisfied in mobile cloud application scenario due to unstable mobile connection to cloud. *Zhang et al.* [7] proposed a web based mobile cloud application model. It defines *weblet* as independent compute unit that provides web service.



(a) PF publishing and discovery.

(b) PF composition.

(c) PF offloading.

Figure 2. POEM functionalities.

Zhang et al. model restricts application structure to User Interface (UI), *weblet* and manifest, which force application components to communicate through web service. MAUI [8] and ThinkAir [9] use similar offloading technique and both have their own decision making algorithms. They require the programmer to mark the offloadable method and they generate two versions of an application: one is for mobile execution and the other is for cloud. Cuckoo [10] focuses on offloading technique. Cuckoo generates local and remote version of an Android Service component, which is similar to MAUI and ThinkAir. Cuckoo requires programmer support to build the final application and it's offloading decision making algorithm is static.

III. Systems and Models

We propose three fundamental POEM functionalities, as shown in Figure 2. There are two POEM frameworks running on two devices or machines. Each POEM framework has an identity so that they can form friendship relation, and the PFs on the framework can benefit from this friendship relation in the service discovery procedure. The items a to e present PFs.

The Figure 2(a) describes how one PF discovers remote available PFs. PF b hosts a service and it publishes the service through local POEM for remote PF to discover. Then PF a can discover the published service on remote side with local POEM PF's help. One prerequisite for a to discover

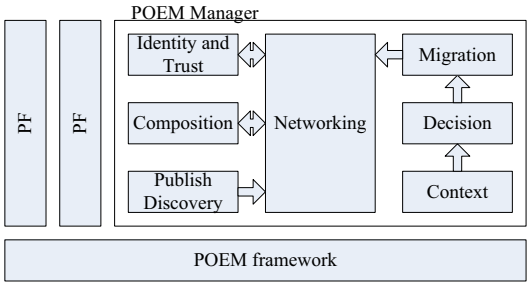


Figure 3. POEM components.

and use service of b is that they are mutual friends, in other words they in each other's contact list. PF a does not know that PF b is running on remote side because POEM pretends that b is running locally. Thus, a programmer does not need special treatments in coding when developing PF a.

The Figure 2(b) presents how an application recruit a service provided by a remote PF. The PF c sends method invocation parameters, which are transferred by the POEM on local side and then on remote side, to the destination PF d. Then, the service result returns along the reverse route from d back to c. PF c also regards it is calling a local target d due to POEM transparent transfer, and d also thinks local c is calling it.

The Figure 2(c) presents how one PF migrates to a remote entity. A POEM PF initializes the migration process. There are two types of migrations: pull and push. In pull migration, the POEM PF on the right side sends request to left side POEM PF, and then the later fetches and transfers the target PF e to the right side. In push migration, the POEM PF on the left side transfers PF e to the remote side. The source keeps the PF e active during transfer to provide the failsafe when the transferring is not successful.

IV. POEM Design

POEM is designed for a distributed application running platform and provides service publish, discovery and composition as a uniform execution environment. In this environment, transparent and seamless PF migration is the key POEM function, i.e., mobile users will not notice the platform level operations when running POEM supported applications.

Figure 3 illustrates the overall design of POEM system. The POEM Manager monitors local services, tracks service state change, maintains local PF repository and responds to remote service queries. Its networking component also maintains XMPP connections to XMPP peers that provides the communication and signaling infrastructure among mobile devices and their ESSIs. The POEM composition component creates local proxy for remote service provider that responds to service request by transferring the request to the remote PF, and then getting the result to the local PFs. Based on a systematic decision model, POEM initiates the migration operations for PF offloading. In the following sections, we describe each component within the POEM framework.

A. Distributed POEM Service Platform

POEM's networking and signaling system is deployed based on XMPP approaches. The communication between POEM entities (i.e., mobile devices and ESSIs) is full duplex compared to half duplex HTTP approach deployed by many web-based service frameworks. In a distributed execution environment, any entity can be both a client and a server at the same time, which is different from web-based service models where clients and servers are explicitly defined. Moreover, POEM inherits the XMPP trust and identity management framework, where every POEM entity is authenticated when joining the system and data transferred are also protected through cryptographic approaches. As a result, the PF offloading and PF compositions can utilize the XMPP trust management framework with fine-grained access control capabilities. Furthermore, POEM entities need to provide their presence information to indicate its availability information in real-time, which is as well used to indicate their service status.

1) POEM Service Discovery and Publishing: POEM service discovery is designed based on XMPP service discovery protocol[11] and XMPP publish-subscribe extension[12]. A PF may reside on a mobile device or its corresponding ESSI. The ESSI takes the responsibility to represent the mobile user for any PF related operations and the mobile device POEM Manager can frequently update its available PFs information to the ESSI. In this way, the main POEM service discovery, migration, and composition operations will not be flooded to end mobile devices. The ESSI POEM Manager also maintains the mobile device availability information and provides its reachability information to its trusted POEM peers. When the ESSI POEM Manager receives the service discovery message, it replies with its available PFs with the available remote service interfaces.

POEM Manager also monitors local service changes and notifies its friends. This is done through a publishing procedure. POEM Manager first registers a publish node (i.e., a virtual node in the XMPP server) under its JID. Thus, when local service status changes, POEM Manager can post the notice on its publish node and its friends get notified and update their PFs availability database. We note that this concept can be extended to the scenario for POEM users are not on each others friend list. POEM can create interest groups for who have registered and receive notices published in the corresponding interest group.

2) POEM Service Composition: When POEM discovers service provided by remote POEM entities, it tries to create a proxy for that service so that remote PF can be used locally. POEM uses Java dynamic proxy technique to create proxy. Dynamic proxy requires that the target interface's

Class instance must exist. To have remote service interface's Class instance in local OSGi framework instance, POEM fetches PF JAR file corresponding to the target service from remote POEM framework. POEM Manager installs the PF, and then the target Class instance is available and proxy generation is done.

POEM uses JavaScript Object Notation (JSON) over XMPP for service composition because JSON is lightweight and has abundant expression ability. The service proxy generated by POEM Manager captures local service requests that are then converted into JSON requests. Then the JSON request is sent to XMPP channel to the destination. The destination POEM Manager receives the JSON request and translates it to method invocation on service provider's object. It then returns the result in form of JSON back to the source POEM Manager. Then the JSON response is decoded and returned to calling object.

B. PF Offloading

When application decides to offload a service provider object and migrate it to cloud, POEM Manager chooses to send the object's byte code to cloud and start the object from byte code. How to choose POEM PFs to be migration is based on several conditions described as follows: First, thread migration solution is not adopted because some objects that exist in the same thread have to run on mobile device, such as user interfaces and sensors. Second, an application usually wants to migrate only the compute intensive operations rather than the whole thread. Third, object state is not maintained because the insight private details of the object to be migrated cannot be fetched due to Java security management. Our recent practice suggests that service implementation should be stateless, so that the object states will not bother POEM like Representational State Transfer (REST) does [13].

1) Migration: The service provider object offloading process follows a three-step approach: First, the target PF JAR file is transferred to ESSI and started. Then, a proxy object is created to intercept and capture service request to remote target service. Finally, the PF containing target service provider object is stopped.

The migration happens according to the migration decision module command. POEM constructs the migration decision module as plug-in framework. User can develop the migration decision strategy plug-ins and install the strategy bundle into POEM, which not only provides the flexibility for user customized migration strategy but also scales the POEM intelligence.

2) PF Isolation: The migrated PFs are running in the surrogate POEM framework for providing service for its origination. These PFs may interact with the POEM framework and interrupt the PFs that belong to surrogate host. The PF isolation is required to protect the surrogate POEM framework and cease the potential attack from the migrated PF.

The POEM manager initializes a separate PF container for each friend who wants to offload his PF. The PF container is duplication of the surrogate host POEM framework. The only difference is that this nested PF container is empty and dedicate for the corresponding friend. The friend identity is stored and managed by identity manager. The surrogate host defines the accepted PF policies that are enforced by policy manager.

3) Connection Failsafe: The connection between mobile device and cloud is usually not stable as mobile device moves. When the connection is lost, POEM Manager restarts the PF that has been stopped in offloading process. The recovery process has the following two steps: First, the target PF is started. Then, the proxy service is unregistered and the proxy object is destroyed. The first step prepares for receiving service request. The second step destroys proxy, which makes the target service provider object be the first in the ranking order to receive service request.

V. POEM Implementation

This section describes the implementation details of the POEM Manager OSGi bundle as well as the seamless offloading procedure.

A. POEM Manager OSGi Bundle Implementation

POEM Manager consists of several objects as shown in Figure 4. They are categorized as three sets - XMPP connection and related listeners, PF context and related listeners, and proxy and migration management. The three object sets represent three POEM functional sets: XMPP connection set represents remote POEM framework; PF context set represents local POEM framework; and proxy and migration management represent core POEM logic and operation that connect the other two parts.

XMPP connection object maintains three XMPP managers that manage service discovery, publish-subscribe, and file transfer separately. Besides, it also maintains a *roster* that publishes local presence and a *publish node* that local service change notification is posted on. There is a set of listeners registered with XMPP connection. They are noticed when corresponding events occur. *Roster listener* tracks friends' presence and update proxy pool accordingly. *Item event listeners*, one listener for one friend, wait for friends' service change notice and update proxy pool accordingly. *Connection listener* monitors connection status and executes robustness strategy. *File transfer listener* handles file transferring. *Packet listeners* handle *iq* packets defined in POEM name space between POEM PFs. *Service discovery provider* responses to remote service discovery by querying PF context.

Other POEM components are as follows: *PF context* handles interaction to POEM framework. *Service listener*

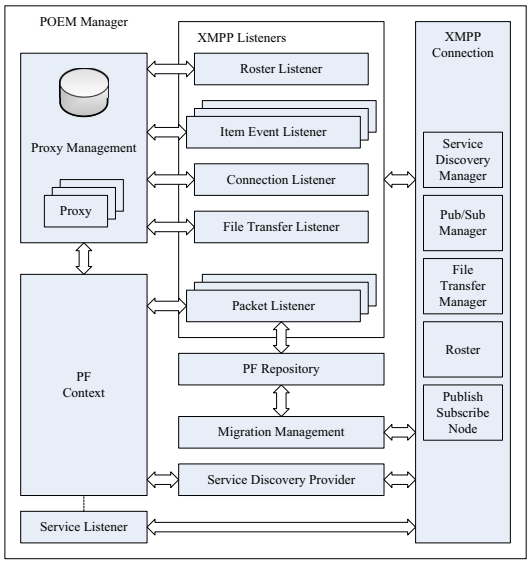


Figure 4. POEM Manager Details

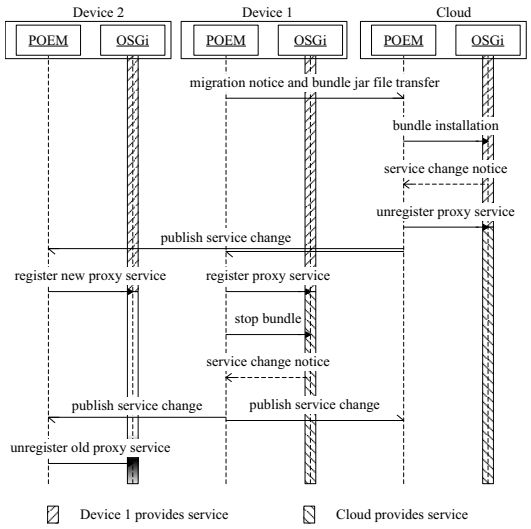


Figure 5. POEM Migration Sequence

monitors local service change and publish change to publish node maintained by XMPP connection. *Proxy management* contains a database and a *proxy* pool. It memorizes remote service status and local proxy status in database, and provides proxy generation and recycling methods. *Migration management* implements migration service registered by POEM Manager. *PF repository* provides JAR file source for file transfer request.

B. Seamless Offloading

POEM Manager registers a service with an Java interface that contains a method to do service migration. Service migration involves two framework instances that are source framework and destination framework. The offloading process can be illustrated using the following application scenario. The source is device 1 and the destination is an ESSI. The migration method is called on device 1. Service name and destination XMPP identity are passed to the migration method. The migration process consists of five steps as follow. First, a migration notice is sent by device 1 to the ESSI. Along with the migration notice, the PF JAR file that owns the indicated service is transferred from device 1 to the ESSI. Second, POEM Manager in the ESSI starts the PF. When PF is running, services including the indicated service are registered. Third, POEM Manager in the ESSI is notified with service changes in last step. it unregister existing proxy under the same service name. Then it publishes the new services to the ESSI's publish node. At this point, both sides have the running PF that provides services to local PFs. Fourth, POEM Manager on device 1 is notified due to the publishing in last step. it creates the proxy for the published services with a higher ranking. Then it stops the local PF. At this point, the PFs on device 1 are consuming services provided by the ESSI. The sequence diagram of migration process is shown in Figure 5.

Besides device 1 and the ESSI, a third framework instance on device 2 is using the service being migrated. When POEM Manager in the ESSI signals the new service, POEM Manager on device 2 creates proxy for the new service with a higher ranking as device 1 does. When POEM Manager in the ESSI signals the service recycling, POEM Manager on device 2 recycles the proxy for that service.

VI. Performance Evaluation

This section describes POEM performance evaluation and case study.

A. Methodology

The POEM Manager is implemented on Felix [14] OSGi implementation version 4.0.3. Mobile application that contains a Felix OSGi framework instance that hosts POEM Manager runs on Android Motorola phone A855. The phone's parameters are 600MHz CPU and 256M memory. The Android version is 2.2.3. The virtual machine is with 1GHZ CPU and 512M memory, which runs Ubuntu 11.10.

Four applications are used to evaluate the POEM performance. They are Fibonacci sequence generator, N-Queens puzzle, nested loop and permutation generator. The Fibonacci application generates Fibonacci sequence in a recursive manner. Its time complexity is $O(2^n)$ and its stack usage is high due to recursive algorithm. The N-Queens application calculates all solutions for input chessboard size. Its time complexity is $O(n^2)$ and its stack usage is also high due to recursive algorithm. The nested loop application contains a six layer loop which leads to time complexity $O(n^6)$. The permutation application's time complexity is $O(n!)$ and uses little memory. Experiment result is obtained by running

Table I
MAX SPEED UP

Case	Input	Phone (ms)	Cloud (ms)	Max speed up (ms)
Fibonacci	26	59.25	2	57.25
	27	99.5	3.05	96.45
	28	156.75	5	151.75
	29	251	7.65	243.35
	30	408.25	12	396.25
N-Queens	8	11	1.1	9.9
	9	39.75	3.05	36.7
	10	222.75	12.2	210.55
	11	1593.5	64.4	1529.1
	12	9630.25	377.2	9253.05
Nested loop	14	157	15.05	141.95
	15	332	21.55	310.45
	16	276.75	28.6	248.15
	17	392.5	39.85	352.65
	18	560.25	54.35	505.9
Permutation	5	1.25	0.25	1
	6	1	0.25	0.75
	7	6.5	0.4	6.1
	8	49.25	2.05	47.2
	9	1124.75	12.1	1114.65

the application 50 times for every scenario and averaged. Between two consecutive executions there is a pause of 1 second.

The experiments are run under two scenarios:

- Phone: Applications are run only in phone.
- WiFi: Phone is connected to the ESSI through WiFi.

The WiFi connection has averaged latency of 70 ms, download bandwidth of 7 Mbps, and upload bandwidth of 0.9 Mbps. Ping is used to report the average latency from the phone to the ESSI, and Xtremelabs Speedtest, downloaded from Android market, is used to measure download and upload bandwidth.

B. Macro-benchmarks

For typical input parameter values, four applications are run on phone and in the ESSI separately. The application running time is recorded in Table I. By subtracting time on phone and in the ESSI, the max speed up is put in the last column of the table. However, the max speed up is seldom achieved due to cost of communication and proxy. This cost changes little while offloading benefit changes much, so there should be some point when the benefit of offloading surpasses its cost giving application net gain.

Fibonacci application takes a sequence index number and calculates the corresponding number in the Fibonacci sequence. Figure 6(a) shows execution time of Fibonacci application. The intersection of execution time on phone and WiFi offloading is the Boundary input value (BIV) [9] that shows the offloading benefit starting point. N-Queens application takes chess board size and calculates all solutions and return solution number. Figure 6(b) shows execution time of N-Queens application. The execution time on phone rises dramatically as the chessboard size increases one scale. Offloading offers benefit after chess size is larger than 10. Nested loop application takes loop times and execute loop without memory operation. The execution time on phone is convex, which means it is less than exponential increase compared to the above two applications that requires both computing and storage. The execution time of offloading increases slowly. The Permutation application takes a max number N and returns count of prime number within the range (1,N). The prime number searching algorithm used is Permutation algorithm. The execution time increases on phone, however the execution time for offloading approach almost remains same.

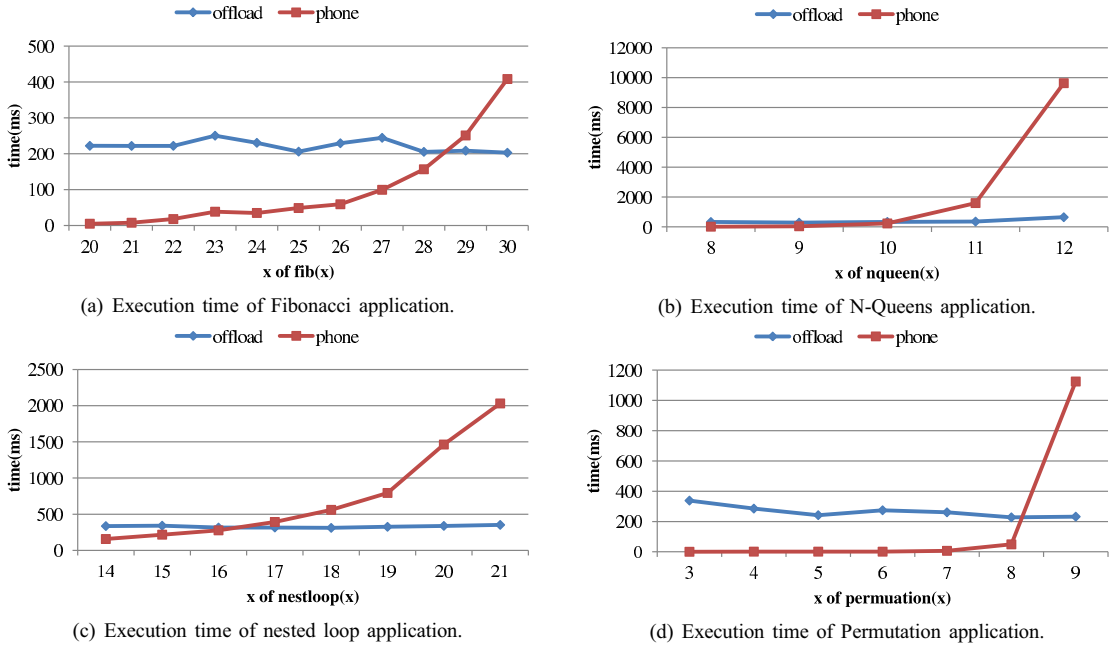
The offloading line of four applications is increasing slowly compared to phone line. As the phone line starts from a low point, which indicates the application runs fast when input is small, the offloading line and phone line intersects finally. Comparing offloading line and the ESSI execution time column in Table I, the slow increase is reasonable due to execution time increase slowing in the ESSI as well. Besides, the starting point of offloading line is higher than phone line, so there must be cost for remote method invocation.

C. Micro-benchmarks

This experiment measures service invocation time. This time is measured on phone where is service consumer side. The remote service consuming time consists of three parts: marshaling time of both consumer and provider sides, network transfer time and actual execution time. The result is shown in Figure 7.

Figure 7 shows time against different input parameters. From the table, the actual execution time is similar to the execution in the ESSI of column the ESSI in table I. At the beginning, execution time is nearly zero. The execution time increases along with input parameter value increases. Figure 7 shows that marshaling time is relatively small compared to network delay. Figure 7 also shows that the main cost for remote method invocation is network delay around BIV point. And marshaling time and network time against different input parameters are approximately identical. The marshaling and network cost decides the start points of offloading line in Figures 6(a)-6(d). And execution time decides the trend of those offloading line. If the network delay or the marshaling is reduced in some situation, the offloading line will drop and then BIV point will go to left, which means the range of benefit increase and application components are supposed to be offloaded to the ESSI. In another perspective, if component's ratio of computation cost to network cost increases, it is better to offload that component to the ESSI.

Besides service invocation time, the proxy generation time is also measured. The proxy generation time indicates



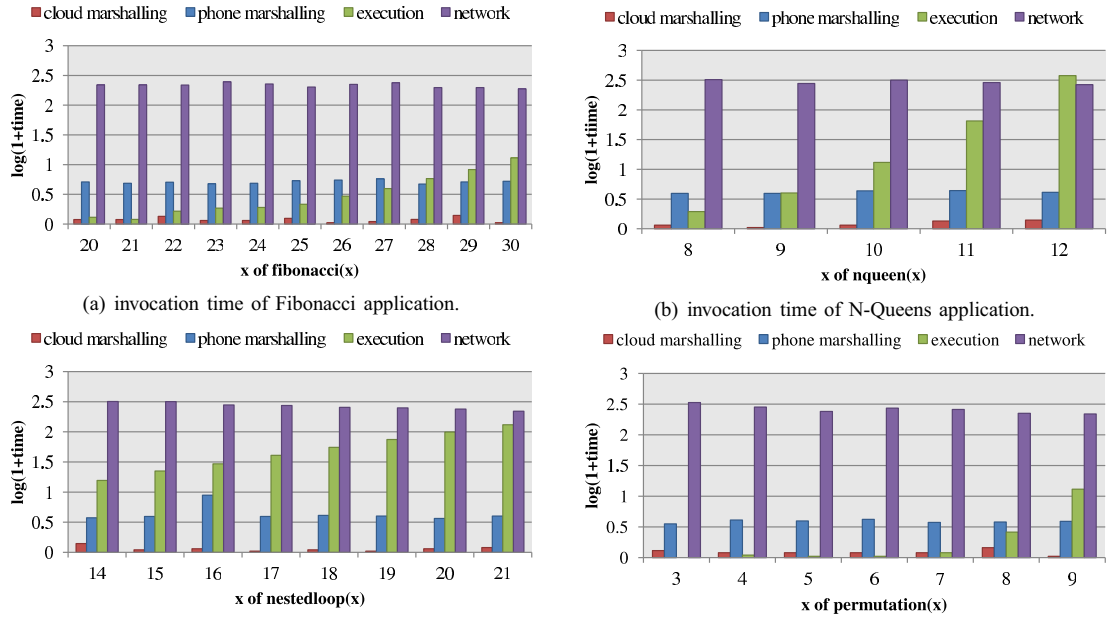
(a) Execution time of Fibonacci application.

(b) Execution time of N-Queens application.

(c) Execution time of nested loop application.

(d) Execution time of Permutation application.

Figure 6. Execution time.



(a) invocation time of Fibonacci application.

(b) invocation time of N-Queens application.

(c) invocation time of nested loop application.

(d) invocation time of Permutation application.

Figure 7. Service invocation time

POEM initialization time, which is paid once at starting POEM Manager.

D. PF Migration

This experiment measures PF migration time. PF migration time period starts when service migration command is issued and ends when proxy for migrated service is available. The result is in table II which shows that the migration time is nearly same for the tested four applications. This is reasonable because the migration time is mainly the time of transferring PF bundles on the network and these four PF bundle sizes are similar.

Table II
SERVICE MIGRATION TIME

Cases	migration time (ms)
Fibonacci	272
N-Queens	335
Nested loop	290
Permutation	304

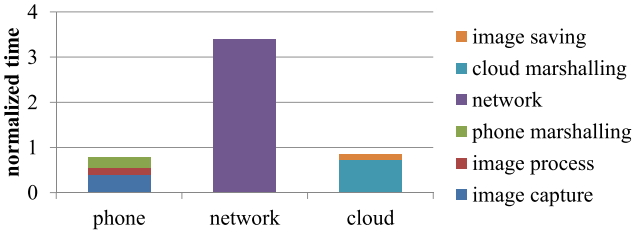


Figure 8. real application evaluation

E. Image Capture Application Evaluation

We developed a remote image capture application to evaluate the prototype. The application implements a PF function to capture the image. The evaluation scenario is that the cloud server composes the image capture function from the remote android phone. The cloud server initiates the PF composition process, and the android phone execute the image capture function and return the image to the cloud server. We measure the time cost for each step in this scenario as shown in Figure 8. In the figure, we use the network handshaking time as the unit time. The phone and the cloud server are connected by a router that is also wifi access point for the phone. From the figure, we can see that the time spent for the POEM prototype is relatively lower than the time for image transfer, which shows the good performance of the prototype.

VII. CONCLUSION

This paper proposes a novel application running platform for mobile cloud computing that allow mobile users to offload and compose mobile cloud application with little management overhead. The implementation is based on OSGi platform and XMPP protocols. The proposed service platform handles service migration, service discovery and service composition seamlessly in a transparent fashion. The evaluation shows the proposed service platform is flexible and efficient. The future work on POEM is to improve security and privacy control of the POEM system. Moreover, the service discover should incorporate more social network features to make the discovery scalable and customizable.

ACKNOWLEDGMENT

The authors would like to thank NSF CPS #1239396 grant to support the research on the MIDAS project.

REFERENCES

[1] *OSGi Core Release 5*, OSGi Alliance, March 2012, http://www.osgi.org/Release5/HomePage.

[2] "Extensible Messaging and Presence Protocol (XMPP), available at http://xmpp.org/," Open Source.

[3] V. March, Y. Gu, E. Leonardi, G. Goh, M. Kirchberg, and B. S. Lee, "ucloud: Towards a new paradigm of rich mobile applications," *8th International Conference on Mobile Web Information Systems (MobiWIS)*, 2011.

[4] R. Ma, K. T. Lam, and C.-L. Wang, "excloud: Transparent runtime support for scaling mobile applications in cloud," in *2011 International Conference on Cloud and Service Computing (CSC)*, 2011, pp. 103–110.

[5] R. Ma, K. Lam, C. Wang, and C. Zhang, "A stack-on-demand execution model for elastic computing," in *Proc. of the 39th International Conference on Parallel Processing (ICPP 2010)*, 2010, pp. 208–217.

[6] 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*, 2011, pp. 301–314.

[7] X. Zhang, S. Jeong, A. Kunjithapatham, and S. Gibbs, "Towards an elastic application model for augmenting computing capabilities of mobile platforms," *Mobile wireless middleware, operating systems, and applications*, pp. 161–174, 2010.

[8] E. Cuervo, A. Balasubramanian, D. 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*. ACM, 2010, pp. 49–62.

[9] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *2012 Proceedings IEEE INFOCOM*, 2012, pp. 945–953.

[10] R. Kemp, N. Palmer, T. Kielmann, and H. Bal, "Cuckoo: a computation offloading framework for smartphones," *Mobile Computing, Applications, and Services*, pp. 59–79, 2012.

[11] J. Hildebrand, P. Millard, R. Eatmon, and P. Saint-Andre, *XEP-0030: Service Discovery*, XMPP Standards Foundation (XSF), 2008, http://xmpp.org/extensions/xep-0030.html.

[12] P. Millard, P. Saint-Andre, and R. Meijer, *XEP-0060: Publish-Subscribe*, XMPP Standards Foundation (XSF), 2010, http://xmpp.org/extensions/xep-0060.html.

[13] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," *ACM Transactions on Internet Technology (TOIT)*, vol. 2, no. 2, pp. 115–150, 2002.

[14] "Apache Felix," http://felix.apache.org/site/index.html, Apache Felix. [Online]. Available: http://felix.apache.org/site/index.html