Outsourcing Resource-Intensive Tasks from Mobile Apps to Clouds: Android and Aneka Integration

Tiago Justino* and Rajkumar Buyya*†
*Cloud Computing and Distributed Systems (CLOUDS) Laboratory
Department of Computing and Information Systems
The University of Melbourne, Australia

†Manjrasoft Pty Ltd, Melbourne, Australia
{tiago.vieira, rbuyya}@unimelb.edu.au

***Abstract*—Mobile Cloud Computing enables augmenting mobile device capabilities and increasing battery lifetime through the extension of cloud services and resources, resulting in an enhanced user experience. However, the development of a mobile cloud application is challenging because it involves dealing with different cloud providers and mobile platforms. To tackle the above issues, a mobile cloud architecture is proposed to asynchronously delegate resource-intensive mobile tasks in order to alleviate the mobile device load and, consequently, extend the battery life. We demonstrate this capability by developing an interface that supports the delegation of heavy tasks from mobile apps running under the Android mobile platform to a cloud computing environment managed by the Aneka Cloud Application Platform. The Aneka Mobile Client Library encapsulates the processes of communicating to cloud is provided, thus, the effort and complexity of developing a mobile cloud application is decreased. Two different resource-intensive mobile application are presented in order to show the library effectiveness. A performance evaluation is conducted showing the feasibility of architecture through the reduction of application execution time and extension of mobile device battery life.**

Keywords*—*Mobile Cloud Computing, Task Delegation, Mobile Apps, Android, Aneka, Software Engineering.

I. Introduction

The International Telecommunication Union (ITU) expects the number of mobile phone accounts to exceed the global population in 2014[1]. This growth is observed throughout all the Smart Mobile Devices (SMDs) domain, such as: Personal Digital Assistants (PDAs), smart phones, and tablets. In recent years, the advance in semiconductor technology enabled the design of mobile devices increasingly powerful and compact [1]. Although the technology has advanced, the miniature and mobile nature of these devices impose intrinsic limitations on CPU, memory, and battery lifetime [2], [3].

This technological advancement enabled the execution of resource-intensive mobile applications, such as voice recognition, image processing, optical character recognizers, and online games. However, SMDs rely on finite energy source and they are resource-poor compared to stationary machines such as desktops and servers [4]. Nevertheless, users presume to execute resource-intensive applications on their SMDs with the same quality expectations (performance and reliability) [3].

[1]http://www.siliconindia.com/magazine_articles/World_to_have_more_cell_phone_accounts_than_people_by_2014-DASD767476836.html

In this context, Mobile Cloud Computing (MCC) enables augmenting SMD capabilities through extension of cloud services and computational resources to SMD on demand [3]. The augmentation of capabilities includes screen, battery life, storage, and application processing (CPU, memory) [5]. Thereby, storage and processing are increased and battery lifetime is extended, enhancing the user experience. Furthermore, this solution inherits characteristics intrinsic to cloud environments like, pay-as-you-go model, elasticity, illusion of infinite resources, and task parallelization [6].

However, the development of a mobile application that requires accessing distributed hybrid clouds is challenging because it involves dealing with different Web APIs from different cloud providers (e.g., Amazon, Microsoft Azure) and different mobile platforms (e.g., Android, iOS, Windows Phone). Moreover, porting these API to SMDs is a difficult task due to compiler limitations, additional dependencies, and code incompatibility reasons [6]. To tackle the above issues, a mobile cloud architecture is proposed to delegate in asynchronous manner resource-intensive mobile tasks in order to alleviate the mobile device load and, consequently, extend the battery life.

We demonstrate this capability by developing an interface that supports the delegation of heavy computing tasks from mobile apps running under the Android mobile platform to a cloud computing environment managed by the Aneka Cloud Application Platform. However, our proposed model for integration of Android and Aneka platforms can be easily applied to other mobile platforms such as iOS and Windows Phone.

The **main contribution** of this paper is the Aneka Mobile Client Library for Android platform that encapsulates the processes of connecting to cloud, serializing and deserializing messages, sending messages, and collecting their responses. Thus, the effort and complexity of developing a mobile cloud application is decreased. In addiction, the library was designed to leverage the Aneka Cloud Application Platform, which provides transparent resource provisioning and job scheduling services and encapsulates different cloud providers Web APIs. The user has no concern in allocating or deallocating virtual machines or distributing the jobs among the resources.

This paper also presents two different resource-intensive mobile applications (ray tracing image generation and Mandelbrot set generation) in order to show the Aneka Mobile Client Library effectiveness. A performance evaluation is conducted

with objective of comparing the applications execution in the mobile device and delegating to the cloud in view of metrics battery consumption and processing time.

The rest of this paper is organized as follow: Section II analyses related works, highlighting similarities and differences for this proposal. Section III describes the architectures and software artifacts, and its deployment. Section IV describes the Aneka Mobile Client Library development and shows its operations. Section V presents two different resource-intensive mobile applications developed using the proposed approach in order to evaluate it. Finally, Section VI concludes the paper and highlights future research directions.

II. Background and Related Work

In order to augment the processing power of mobile devices, two main approaches have emerged [6]: offloading and delegation. In offloading, applications are partitioned into components, which are analyzed in order to determine whether they should be migrated to the cloud. This process can occur at development time or at runtime. These components can be different entities depending on the granularity level, such as method, class, module, application partition, entire application or image [3]. Usually, the analysis considers two variables for deciding about migration: the amount of computation and the amount of data to communicate. A component is migrated if intensive computation and little communication are needed [1]. The finer the granularity, the more intensive is the synchronization mechanism between mobile device and cloud. The more abstract the granularity, the more traffic intensive is the communication [3].

Shiraz et al. [3] describes and compares 17 different works using offloading, considering several aspects such as partitioning approach and migration granularity. Among them we highlight the two following works. Hung et al. [7] propose an Android framework based on VM migration for application offloading. The framework installs an application on the mobile phone for managing the offloading process. This application encapsulates other running applications in VMs to migrate and execute on cloud servers. This approach requires a large amount of bandwidth and processing for ensuring consistency between the mobile phone and the cloud server. Cuervo et al. [8] propose Mobile Assistance Using Infrastructure (MAUI), which adopts a dynamic (runtime) partitioning approach at a method level with focus on energy saving for the mobile device. MAUI leads to extra overhead on mobile devices to analyse the amount of computation needed for each method and to migrate and reintegrate this computation.

Delegation utilizes Service Oriented Computing (SOC) in order to migrate the execution of resource-intensive tasks, e.g. prime verification or matrix multiplication, to cloud. Mobile tasks are delegated by invoking a cloud service [6] and no code is migrated to cloud, what makes this approach more lightweight than offloading. On the other hand, as delegation has dependency on the cloud service to execute part of the computation, it requires always available network connectivity.

In the offloading approach, the cloud utilization is limited to executing applications that can run on mobile devices. On the other hand, delegation approach allows the utilization of resources unavailable in the mobile device platform, but that are available in the platform running on cloud. Section V discusses an example of this advantage, where the software for image rendering POV-Ray[2], available only for the Windows platform, is used from an Android application. Furthermore, the delegation approach allows the use of other characteristics intrinsic to cloud environments, such as dynamic resource provisioning, elasticity, illusion of infinite resource, and task parallelization [6], as presented in this work.

Abolfazli et al. [2] implement mobile augmentation by utilizing delegation and analyses how the number of hops impacts the execution time in a service call. They prefer SOC instead of offloading in order to avoid the overhead of identifying, partitioning and transferring large amounts of data from mobile to cloud. They do not leverage cloud aspects like dynamic provisioning or parallel tasks execution.

Flores and Srirama [6] propose a Mobile Cloud Middleware (MCM) to work as an intermediary between the mobile phone and the cloud in order to manage the asynchronous delegation of mobile tasks to cloud resources. MCM abstracts the API and manages different cloud providers, as well as allows the development of customized services based on service composition. The mobile task computation happens on the cloud providers and a connection between MCM and the providers is kept during all the execution. After the computation is finished, MCM stores the results in the transactional space and sends a message to the mobile application via push notification, signalling the end of execution.

Although MCM communicates with the cloud resources, the mobile application is responsible for managing the information about the services and the cloud providers, which increases the mobile application complexity and reduces flexibility. If any piece of this information changes, the mobile application needs to be updated. In addition, MCM requires services to be previously developed and deployed. In this work, the Aneka cloud provides a dynamic resource provisioning allowing scale up and down according to application needs, which is not observed in MCM.

III. Architecture

This section describes the elements that compose the proposed solution architecture and how these elements relate as shown in Figure 1. The deployment of the architecture is presented in Figure 2. The two main structures in the architecture are: (i) Aneka service, which provides via web interface a runtime environment for tasks execution; and (ii) mobile client, which allows mobile applications to send resource-intensive tasks to execute on cloud and alleviate the mobile device load.

The architecture was designed in order to facilitate the development and deployment of mobile cloud applications. Firstly, by using the *Aneka Mobile Client Library*, all the complexity of communicating to Aneka cloud and submitting jobs is taken care for the developer. Secondly, by using the Aneka cloud, which provides transparently the resource provisioning and job scheduling services, the user has no concern in allocating or deallocating virtual machines or distributing the jobs among the resources.

[2]http://www.povray.org/

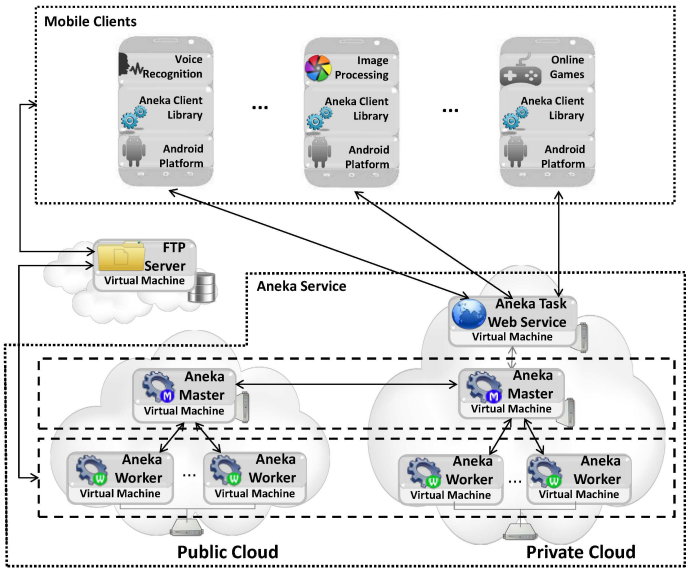


Fig. 1. Architecture showing how mobile devices interact with an Aneka cloud through the *Aneka Task Web Service* in order to outsource resource-intensive tasks.

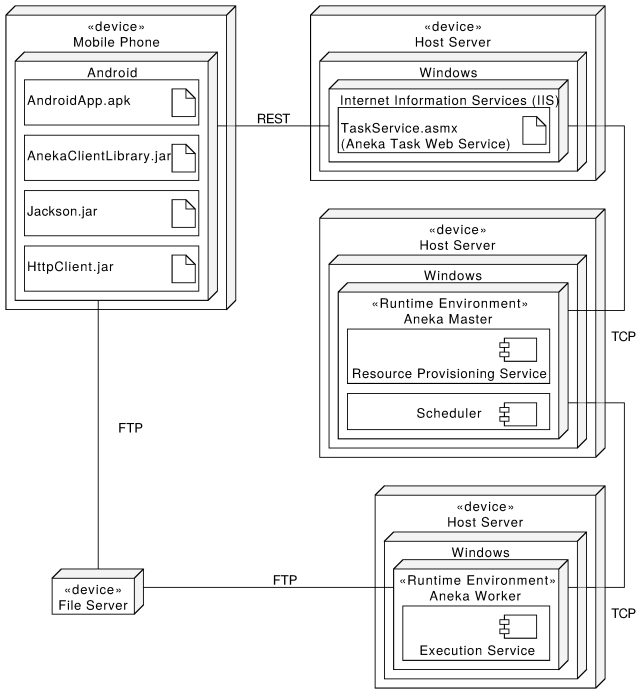


Fig. 2. Deployment Diagram showing how the different artifacts can be deployed and how they communicate.

A. Aneka Service

Aneka [9] is a PaaS solution for developing cloud applications that can be deployed on both public and private clouds. It provides a runtime environment as well as an Application Programming Interfaces (APIs) to build .NET applications leveraging the parallel power of an Aneka cloud. By using these APIs, developers can implement and deploy applications that automatically scale on demand, following different programming models such as Bag of Tasks (BoT), Thread, and MapReduce.

An Aneka cloud is defined as a collection of physical resources (desktops, servers) and/or virtualized resources (virtual machines) connected though a network, each of these machines running an instance of Aneka Container. This container represents the basic deployment unit of Aneka and it provides a runtime environment composed of *Aneka Master* and *Aneka Workers* to execute the distributed applications.

Aneka is designed following a Service Oriented Architecture (SOA), which makes it customizable and extensible, allowing developers to create new services that replace the default ones or that add new functionalities to Aneka. The default installation provides services such as dynamic resource provisioning, resource reservation, persistence, storage, security and performance monitoring [9].

The *Aneka Master* hosts the dynamic resource provisioning service, which is responsible for dynamically acquiring and integrating new *Aneka Workers* into the Aneka cloud, allowing it to elastically scale up and down to satisfy the applications needs. It also hosts the *Scheduling Service*, which is responsible for dispatching the collection of application jobs to the *Aneka Workers*, as shown in Figure 2. Thereby, when a mobile cloud application is executed, its jobs are submitted to the *Aneka Master*, via *Aneka Task Web Service*. Each of these jobs is moved to *Aneka Workers* and processed by the *Execution Service*, which is the runtime environment in charge of retrieving all the files required for execution, monitoring job execution, and collecting results.

The *Aneka Task Web Service* enables applications developed in any language to send mobile jobs to run on Aneka cloud [10]. This service exposes, via Representational State Transfer (REST), the main functionalities such as authenticate user, create application, submit jobs, and query information (see Figure 2). REST was preferred to SOAP because the message serialization process is faster [11], configuring an advantage to *Aneka Task Web Service*.

B. Mobile Client

The mobile client side of the architecture contains Mobile Cloud Applications, *Aneka Mobile Client Library*, and *Android Platform* layers, as presented in Figure 1. Mobile cloud applications, such as voice recognition, image processing, optical character recognizers, and online games, benefit from cloud computing resources to execute resource-intensive tasks.

Depending on the mobile application characteristics, input files may need to be uploaded to the cloud. For instance, an audio file needs to be uploaded for a voice recognition application or, a image file for an image filtering application. In order to keep a light implementation, the web service interface was designed not to allow user to upload or download files through it. An easy way to transfer files to use in Aneka cloud is through a storage service. As shown in Figure 2, the FTP is used to communicate with the *File Server*.

Once the mobile application developer identifies a resource-intensive task, they can instantiate the *Android Client Library* in order to consume the *Aneka Task Web Service*. The *Android Client Library* hides the complexity of delegating tasks to Aneka, executing steps, such as connecting to the

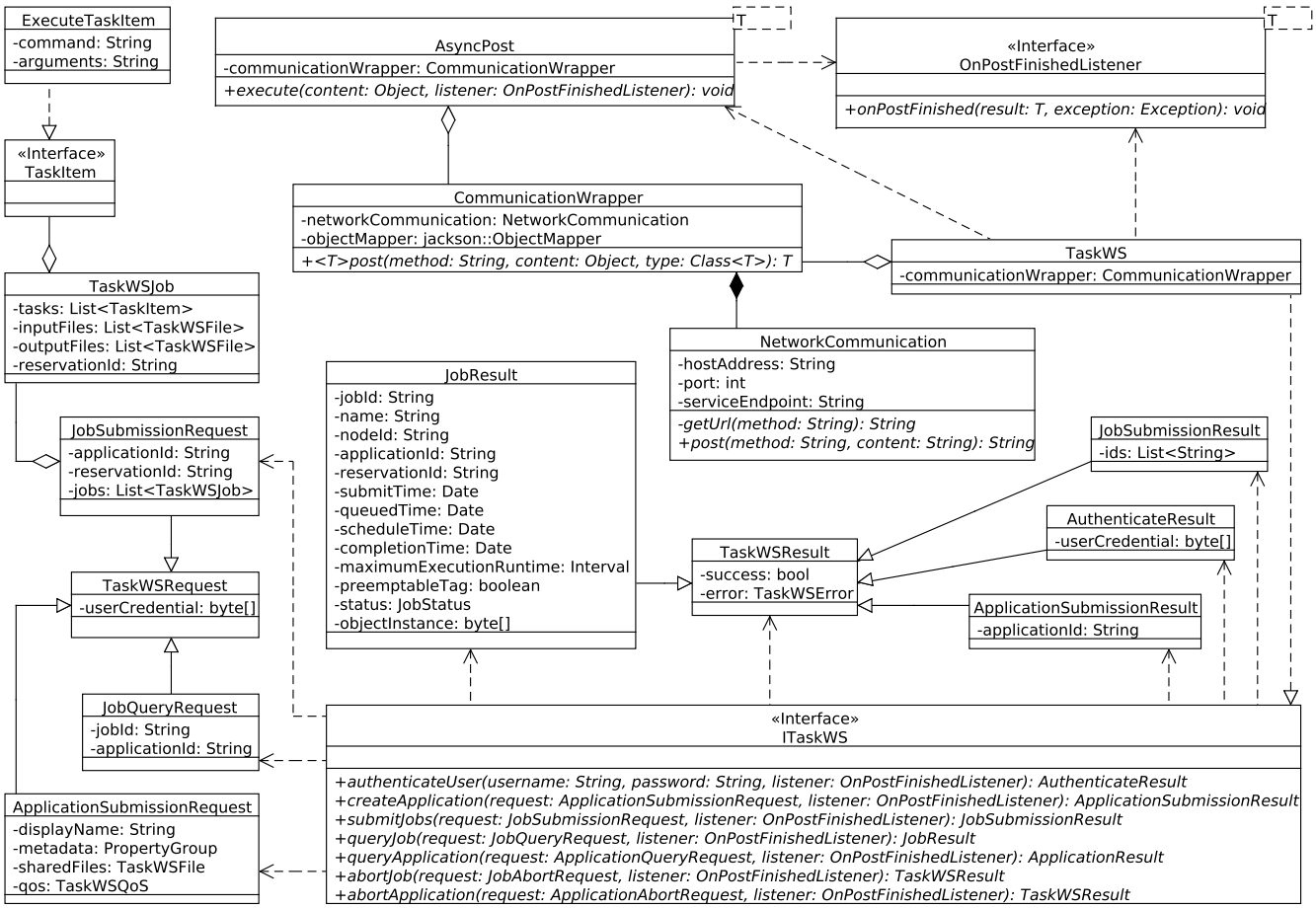


Fig. 3. Class Diagram showing the main classes and interfaces, such as *TaskWS* and *ITaskWS*, and how they interact.

server, serializing and encapsulating objects into requests, sending messages, and collecting their responses. Thus, the mobile cloud application development is simplified since the library can be easily reused by new applications.

The serialization and deserialization process of messages exchanged between the *Android Client Library* and the *Aneka Task Web Service* is provided by the third-party library called Jackson[3]. The messages are serialized to JSON format and sent to *Aneka Task Web Service* through the Http-Client class, provided by Apache[4]. The JSON format simplifies the communication between different platforms. Additionally, it is more lightweight in comparison with XML [11].

The *Android Client Library* invokes the Aneka service in an asynchronous manner since the resource-intensive task requires time to process and keeping the connection entails mobile device battery consumption. Also, these devices are prone to connection loss due to their mobility characteristic, which makes the synchronous communication an unwise choice. So, in the event of connection loss during a synchronous task execution the information gets lost and the message needs to be retransmitted, entailing consumption of time, mobile device battery, and cloud resources.

[3]http://jackson.codehaus.org/

[4]http://hc.apache.org/httpcomponents-client-ga/

The *Android Client Library* was developed in Java in order to execute over the *Android Platform*. However, the library could be developed for iOS or Windows Phone, moreover, the architecture does not depend on the mobile phone platform.

IV. Design and Implementation

This section details the design and implementation of the Android Client Library proposed in this work. This library cover all 7 operations provided by the *Aneka Task Web Service*, which are: user authentication, application creation, application query, application abortion, job submission, job query, and job abortion.

Figure 3 shows the main library classes and interfaces. The *ITaskWS* interface holds the *Aneka Task Web Service* methods. This interface is implemented by *TaskWS* class. *TaskWS* is responsible for creating and storing a *CommWrapper* instance. *CommWrapper* has only one method, called *post*, implemented using Java Generics. This method receives as parameters a string containing the web service operation to be executed, the content to be sent to the web service, and return type for which the response will be converted.

TaskWS also instantiate a *AsyncPost* object every time it needs to execute an asynchronous operation with the web service. *TaskWS* transfer its *CommWrapper* instance to *AsyncPost*

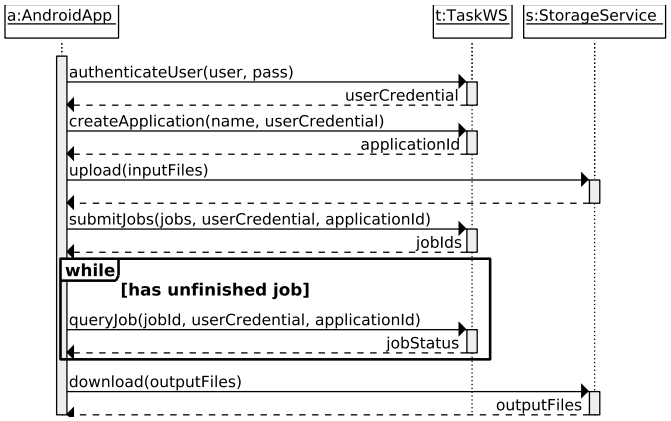


Fig. 4. Basic lifecycle for a mobile application using Aneka.

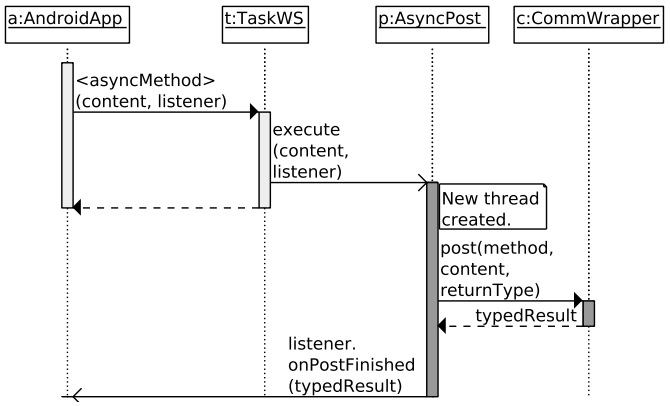


Fig. 5. Sequence diagram for asynchronous methods.

through the constructor. *AsyncPost* implements the *AsyncTask* interface from Android API, which is responsible for creating thread for executing the *CommWrapper post* method. *CommWrapper* uses the *NetComm* class, which is responsible for generating the HTTP message, assembling the URL for HTTP call and sending the message to the web service through the apache *HttpClient* class. *TaskWS* needs also to give *AsyncPost* an *OnPostFinishedListener* instance. This instance will be called by *AsyncPost* after *CommWrapper* finishes its *post* operation.

A. Job Submission

Mobile applications using *Aneka Mobile Client Library* will commonly follow the lifecycle described by Figure 4 to submit jobs to *Aneka Master* through *Aneka Mobile Client Library*. The main library element is the *TaskWS* class which encapsulates all the communication with *Aneka Task Web Service*. Thus, a mobile application, represented in the figure by the class *AndroidApp*, starts the job submission process by using the *authenticateUser* methods. This method returns the user credentials, which will be required by all the other *TaskWS* methods. After authenticating the user, a mobile application creates the application entity in the Aneka cloud via *createApplication* method, which returns the application id. The FTP server is represented in Figure 4 by the *StorageService* class. The file transfer is done through the *upload* method.

Following the *createAppplication* method, and input files upload if needed, the mobile application is able to submit jobs by using the method *submitJobs*, which returns the job id for each successfully submitted job. This identifier is used to query jobs status via *queryJob* method during their execution. Once the job execution has finished, the *AndroidApp* can download the output files, e.g. a text file for a voice recognition application or a new image for an image filtering one.

B. Asynchronicity

Each of TaskWS's method is accessed in an asynchronous way. The library was purposefully designed this way to facilitate the developers work with Android, as it does not allow network I/O operations in the main thread [5]. This restriction is imposed because network I/O operations in the main thread entails blocking the user interface. In this context, we developed the *AsyncPost* class extending the *AsyncTask* class provided by Android. The latter instantiate a new thread, releasing the application to continue its flow.

Figure 5 shows the sequence diagram for a method called in an asynchronous way. In order to call a method from *TaskWS* class, e.g. *authenticateUser*, *AndroisApp* needs to input a content, e.g. user name and password, and a listener. This input is forwarded to the *AsyncPost* class, that creates a new thread and calls the *post* method from *CommWrapper*. This method encapsulates the network I/O operations and returns the method result to *AsyncPost*. In the case of user authenticate example, it returns the user credential. Finally, the *AsyncPost* calls the listener delivering the result to *AndroidApp*.

Figure 6 details how the exchange of messages between the *Aneka Mobile Client Library* and the *Aneka Task Web Service* is performed. The *CommWrapper* is responsible for encapsulating the process of serializing and deserializing the messages via the *Mapper* class provided by Jackson library as well as communicating to the remote service through the *NetComm* class. Thereby, when the *CommWrapper*'s *post* method is called, it receives a Java object as parameter. This object is serialized and sent to *NetComm*.

The *NetComm* class is responsible for assembling the Uniform Resource Locator (URL), generating the POST request message, and sending it through the *HttpClient* class. When the network operation finishes, the result is returned to *CommWrapper*, which uses again the *Mapper* class in order to deserialize the JSON content to Java object and returns it to *AsyncPost* class.

V. Use Cases and Performance Evaluation

This section shows the Aneka Mobile Client Library's effectiveness through the development of two resource-intensive Android applications, one for image rendering called DroidPov and one for generating the Mandelbrot set called MandelDroid. Furthermore, this section evaluates the MandelDroid application in order to quantify the gain by using the cloud resources in terms of execution time and battery consumption.

[5] http://developer.android.com/reference/android/os/NetworkOnMainThreadException.html

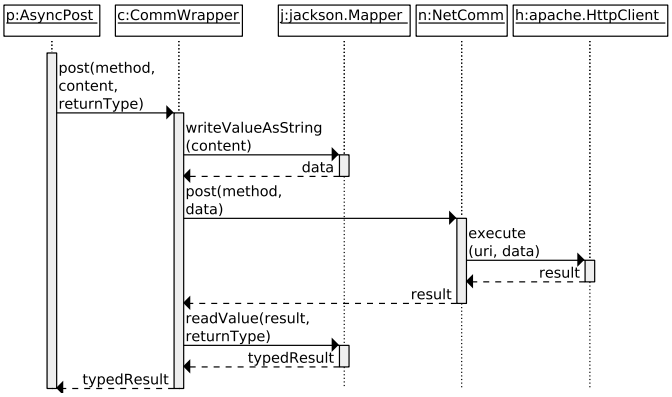
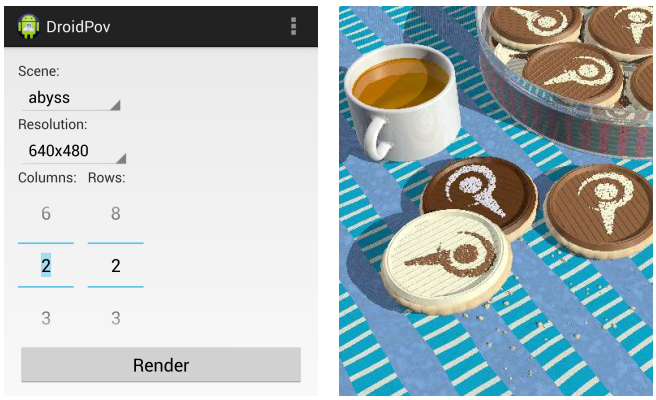
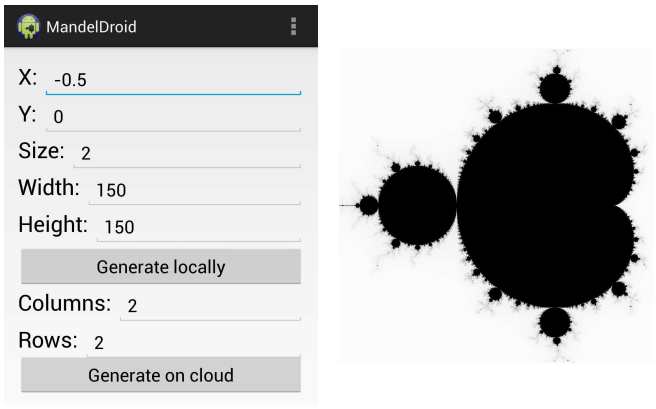


Fig. 6. Sequence diagram for *CommunicationWrapper* class *post* method.



(a) DroidPov *Main Activity*. (b) Scene generated image.

Fig. 7. DroidPov Screens showing (a) the user input parameters: the *Resolution* and the *Scene* to be generated and the number of *Columns* and *Rows* to split the image generation; and (b) the generated image downloaded to the mobile phone.



(a) MandelDroid *Main Activity*. (b) Mandelbrot set generated image.

Fig. 8. MandelDroid screens showing (a) the user input parameters: *X* and *Y* representing the central point, the *Size* of the set, and *Width* and *Height* of the image to be generated; and (b) Mandelbrot set generated image shown in the mobile phone.

A. Use Cases

Alrokayan and Buyya [12] show how the use cases presented in this section benefit from the programming models supported by the Aneka Cloud Application Platform. However, in this section, the same applications are ported to run on Android mobile phones.

DroidPov uses the Persistence of Vision Ray-Tracer (POV-Ray)[6] tool for generating images through the ray-tracing technique [13] from a text file which describes a scene, defining aspects such as light, objects, camera position and atmosphere effect. This file is stored on the mobile phone and sent to Aneka cloud through a FTP server. This type of application has a resource-intensive nature [14]. The delegation approach enables the mobile platform to use the POV-Ray software, available only for Windows platform. Thereby, this approach adds the advantage of allowing the utilization of resources unavailable to the mobile device platform, but that are available in the platform running in the cloud.

As shown in Figure 7a, in order to use the DroidPov, the application user has to set the scene to be rendered, the generated image resolution, and the number of columns and rows to generate the image in parallel. The product of number of rows and columns defines the number of jobs that will be processed by Aneka Workers. Once the image processing has been completed, the image is downloaded and rendered on the mobile phone screen as shown in Figure 7b.

The other developed application is MandelDroid, which consists of a mobile application to generate the Mandelbrot set. This application receives as input a range in the plane of complex numbers. This range is defined by the origin point coordinates and its size, as illustrated in the Figure 8a. The user also specifies the generated image resolution and, if the application runs in the cloud, the number of columns and rows that define how many jobs demanded to split the image generation.

The Mandelbrot algorithm assesses how quickly each point belonging to the range converges to infinity. A gray-scale color is assigned to each point, as shown in Figure 8b. The black points do not converge to infinity, therefore, they belong to the Mandelbrot set. On the other hand, the white points converge to infinity so they do not belong to the Mandelbrot set.

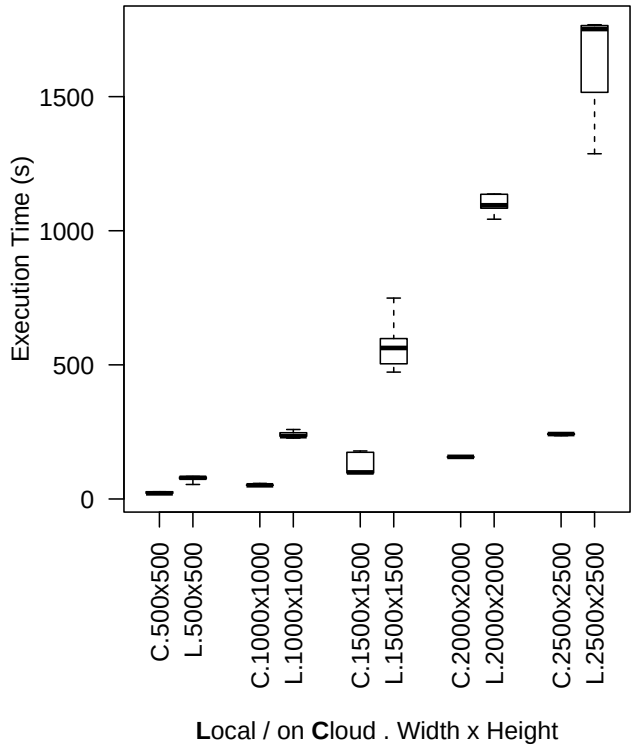
In both applications, the user must configure the connection parameters of Web Aneka Task Service and FTP server, and define a username and password for Aneka authentication.

B. Performance Evaluation

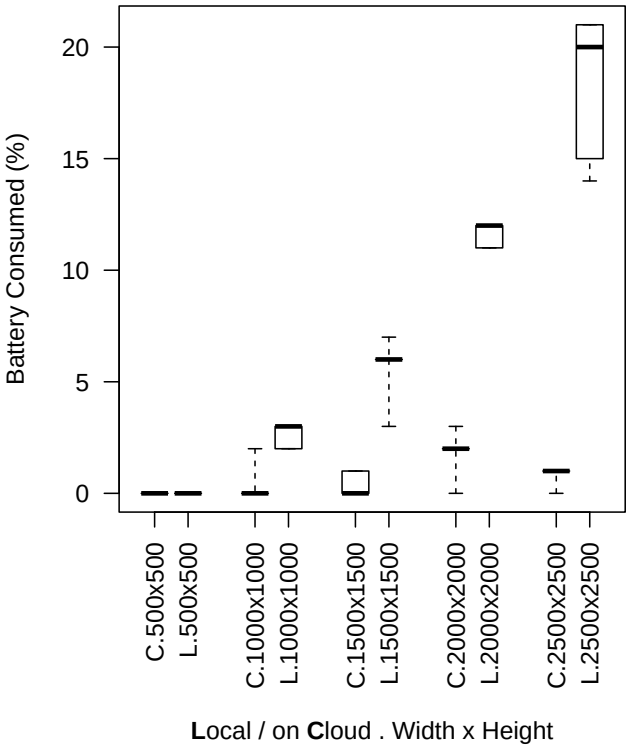
In this section, a performance evaluation is conducted to compare the application execution in a mobile device and in the Aneka cloud and demonstrate the advantage of using the proposed approach, considering the execution time and battery consumption metrics. Since the POV-Ray software is not available for Android platform, only the MandelDroid application is employed in the experiments.

The first experiment consists of executing the MandelDroid application, both on the mobile phone and on the cloud, for generating Mandelbrot set images with 5 different resolutions: 500x500, 1000x1000, 1500x1500, 2000x2000 and 2500x2500. The origin point was fixed in (-0.5, 0) and the range size to 2.

[6]http://www.povray.org/



(a) Execution time, in seconds, for each parameter combination.



(b) Percentage of battery consumed for each parameter combination.

Fig. 9. Boxplot graphs comparing the results for *L*ocal and on *C*loud execution with different image resolutions: 500x500, 1000x1000, 1500x1500, 2000x2000 and 2500x2500. Each combination of parameters was executed 5 times.

Also, the application was split in 4 jobs in order to execute on mobile device and Aneka cloud. Each experiment was executed 5 times, resulting in a total 60 rounds. The observed metrics were battery consumption, in percentage, and execution time, in seconds, both monitored using the Android API.

The experimental scenario is composed of a Asus Padfone Infinity a86 (T004) mobile phone with Android 4.2.2, 32 GB of storage, 2 GB RAM, support WLAN 802.11a/b/g/n/ac and CPU Snapdragon 800 quad-core (2.2GHz) and one Azure Standard tier A3 virtual machine, with 4 2.1GHz cores and 7 GB memory, running Windows Server 2012 R2. The internet connection used to connect the mobile device to FTP server and the Aneka Task Web Service has upload and download rates of 12.11 Mbps and 11.79 Mbps, respectively.

Figure 9a presents the MandelDroid local (L) and on cloud (C) execution time with the different image resolutions. This plot shows that the time spent for the application execution on cloud is lesser for all the resolutions. This difference is bigger as the resolution increases, representing an economy of up to 87% for the 2500x2500 resolution.

Figure 9b presents the battery consumption during the experiment execution, where it can be observed that the consumption showed by the mobile device is bigger for all resolutions except 500x500. This result is expected, whereas during the execution on cloud, the mobile device uses energy only to wait for the remote execution result, for instance, to keep the Wi-Fi connection and display active. For the 500x500 resolution, the required processing to generate the image is low, both using the cloud or not, which represents a negligible impact to battery consumption. The battery savings can reach 95.23% for the 2500x2500 resolution.

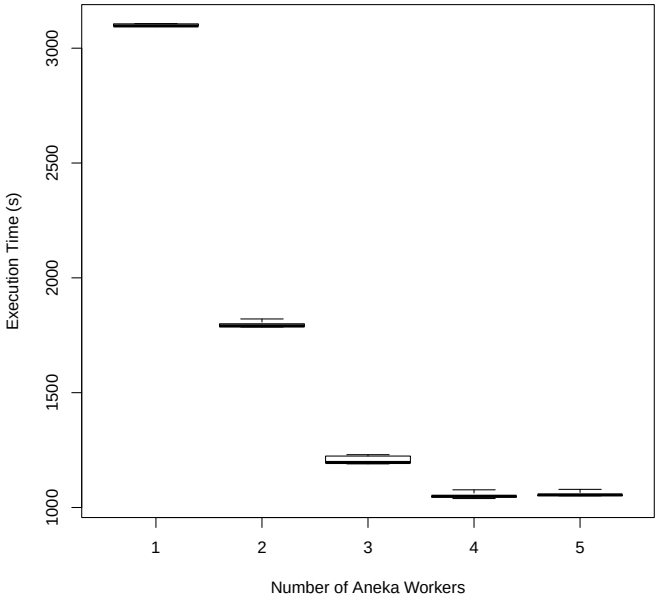


Fig. 10. Boxplot graph comparing the impact of the number of Aneka Workers on execution time. The image resolution is fixed to 12500x12500 and the image generation was split in 25 jobs.

The second experiment aims to evaluate the impact of different numbers of Aneka Workers on the execution time. This

experiment scenario differs from the first because c3.xlarge Amazon EC2 virtual machines were used instead of Azure ones. Each virtual machine has 4 2.8GHz cores and 7.5 GB and runs Windows Server 2008 R2.

Figure 10 presents the execution time for generating a Mandelbrot set image with 12500x12500 resolution. The image generation is split in 25 jobs and distributed among workers that vary from 1 to 5 and each boxplot in the figure corresponds to 5 application execution rounds. For each worker added to the experiment, the total number of CPU cores increases by 4, varying from 4 to 20 cores.

By increasing the number of workers from 1 to 3, the execution time is significantly reduced. However, an expressive gain is not observed when the number of workers increases from 4 to 5. Two different situations can explain this: (i) each of the 25 jobs have similar execution time and at least one core needs to execute 2 jobs; or (ii) the execution time of the jobs are different and the longest jobs are limiting the application execution time. In this context, the ideal number of workers is 4, considering the image resolution and the number of jobs aforementioned and having the reduction of execution time as main goal.

VI. Conclusions and Future Work

Currently, developers are facing complex mobile applications that require accessing distributed clouds. The development of these applications is challenging because it involves dealing with different cloud providers Web APIs and mobile platforms. Moreover, porting these APIs to mobile devices is a difficult task due to compiler limitations, additional dependencies, and code incompatibility. In order to reduce the effort and complexity of developing mobile cloud applications, the Aneka Mobile Client Library was proposed and described in this paper. This library encapsulates the processes of connecting to cloud, serializing and deserializing messages, sending messages, and collecting their responses.

A mobile cloud architecture was also proposed to delegate resource-intensive mobile tasks in an asynchronous manner in order to alleviate the mobile device load and, consequently, extend the battery life. This architecture was designed to leverage the Aneka PaaS solution, which provides transparent resource provisioning and job scheduling services and encapsulates different cloud providers Web APIs. Thereby, the user has no concern in allocating or deallocating virtual machines or distributing the jobs among the resources.

This paper also investigated the effectiveness of the Aneka Mobile Client Library through the development of two resource-intensive mobile applications: ray tracing image generation and Mandelbrot set generation. A performance evaluation was conducted and the results showed the feasibility of the architecture, since the Aneka cloud spends less time to execute the MandelDroid application, representing a reduction of up to 87% of execution time while reducing battery consumption by up to 95.23%.

As future work, we are planning to (i) improve the verification process of job execution by integrating with Android push notification service, avoiding the battery consumption intrinsic to the pooling approach currently used to check the status of submitted jobs; (ii) port the Aneka Mobile Client Library for other mobile platforms such as iOS and Windows Phone; and (iii) design new scheduling and provisioning policies that consider user preferences, such as budget and application execution deadline, and mobile context parameters, such as battery level and internet connection type (WiFi of mobile).

Acknowledgements

This work is supported by the Australian Research Council through Future Fellowship program. We would like to thank Amir Vahid Dastjerdi, Nikolay Grozev, Rodrigo N. Calheiros, Satish Narayana Srirama, and Deborah Magalhães for their comments on improving the quality of the paper.

References

[1] K. Kumar and Y.-H. Lu, "Cloud computing for mobile users: Can offloading computation save energy?" *Computer*, vol. 43, no. 4, pp. 51–56, 2010.

[2] S. Abolfazli, Z. Sanaei, M. Alizadeh, A. Gani, and F. Xia, "An experimental analysis on cloud-based mobile augmentation in mobile cloud computing," *Consumer Electronics, IEEE Transactions on*, vol. 60, no. 1, pp. 146–154, 2014.

[3] M. Shiraz, A. Gani, R. H. Khokhar, and R. Buyya, "A review on distributed application processing frameworks in smart mobile devices for mobile cloud computing," *Communications Surveys & Tutorials, IEEE*, vol. 15, no. 3, pp. 1294–1313, 2013.

[4] M. Satyanarayanan, "Fundamental challenges in mobile computing," in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. ACM, 1996, pp. 1–7.

[5] S. Abolfazli, Z. Sanaei, and A. Gani, "Mobile cloud computing: A review on smartphone augmentation approaches," *arXiv preprint arXiv:1205.0451*, 2012.

[6] H. Flores and S. N. Srirama, "Mobile cloud middleware," *Journal of Systems and Software*, 2013.

[7] S.-H. Hung, T.-W. Kuo, C.-S. Shih, J.-P. Shieh, C.-P. Lee, C.-W. Chang, and J.-W. Wei, "A cloud-based virtualized execution environment for mobile applications," *ZTE Communications*, vol. 9, no. 1, pp. 15–21, 2011.

[8] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM, 2010, pp. 49–62.

[9] C. Vecchiola, X. Chu, and R. Buyya, "Aneka: a software platform for .NET-based cloud computing," *High Speed and Large Scale Scientific Computing*, pp. 267–295, 2009.

[10] R. N. Calheiros, C. Vecchiola, D. Karunamoorthy, and R. Buyya, "The Aneka platform and QoS-driven resource provisioning for elastic applications on hybrid clouds," *Future Generation Computer Systems*, vol. 28, no. 6, pp. 861–870, June 2012.

[11] K. Hameseder, S. Fowler, and A. Peterson, "Performance analysis of ubiquitous web systems for smartphones," in *Performance Evaluation of Computer & Telecommunication Systems (SPECTS), 2011 International Symposium on*. IEEE, 2011, pp. 84–89.

[12] M. Alrokayan and R. Buyya, "A web portal for management of Aneka-based multicloud environments," in *Proceedings of the Eleventh Australasian Symposium on Parallel and Distributed Computing-Volume 140*. Australian Computer Society, Inc., 2013, pp. 49–56.

[13] A. S. Glassner, *An introduction to ray tracing*. Morgan Kaufmann, 1989.

[14] C. Mateos, A. Zunino, M. Hirsch, M. Fernández, and M. Campo, "A software tool for semi-automatic gridification of resource-intensive java bytecodes and its application to ray tracing and sequence alignment," *Advances in Engineering Software*, vol. 42, no. 4, pp. 172–186, 2011.