2013 IEEE Seventh International Symposium on Service-Oriented System Engineering

The Cloud Personal Assistant for Providing Services to Mobile Clients

Michael J. O'Sullivan, Dan Grigoras
Department of Computer Science
University College Cork, Cork, Ireland
{m.osullivan, grigoras}@cs.ucc.ie

Abstract - **This paper introduces the original concept of a cloud personal assistant, a cloud service that manages the access of mobile clients to cloud services. The cloud personal assistant works in the cloud on behalf of its owner: it discovers services, invokes them, stores the results and history, and delivers the results to the mobile user immediately or when the user requests them. Preliminary experimental results that demonstrate the concept are included.**

Keywords: mobile cloud, personal assistant, cloud services

I. INTRODUCTION

The ever-increasing penetration rate of mobile devices such as smartphones and tablets creates opportunities for more flexible and adaptive computing models and applications. Currently, mobile devices allow their users to access remote services on the move. At the same time, clouds provide services on demand. The interaction between clouds and mobiles is already seen as a path to follow for developing new services and applications, as well as overcoming the inherent computing limits of mobile devices. For example, different components of a computing-demanding application can be split between the cloud and the mobile device, optimizing a variety of objective functions [1]. Cloud event notification services can alert subscribers of the status change in some objects of interest (e.g., flights schedule) [2]. These applications show the potential of harnessing cloud resources to the benefit of mobile clients (i.e. mobile users).

According to Mark Beccue, "by 2014, mobile cloud computing will become the leading mobile application development and deployment strategy, displacing today's native and downloadable mobile applications" [3]. Within this model, cloud services are always available with quasi-unlimited resources, relieving the user of administrative tasks. The user chooses the service(s) and pays for what is used. Instead of hosting downloaded apps, the mobile client can avail of existing cloud services and use them as and when they are required. However, this model is not perfect.

Mobile devices can suffer from several problems that can have an adverse effect on their ability to access web-based software and services. They can suffer from a loss in signal, resulting in a disconnection of the device from the mobile network. Wi-Fi is not available everywhere, and where it is available in public locations, a charge is normally required for access. The battery on the device can die as well.

Cloud providers become more aware of the cloud capabilities in terms of services that add value to the model. However, the number of services and subscribers can get to a level where service governance technology is required to manage the system complexity effectively [4].

These aspects require middleware services that will pervasively mediate the access of mobile clients to cloud services and be able to safely and effectively manage the increasing number of users and services.

Our research goal is to go further on the path of integrating clouds and mobiles towards a common, pervasive service space. We are motivated by three key challenges in the areas of cloud and mobile computing: discovery of and access to appropriate services in the clouds (SaaS), the effective provision of services to mobiles in all situations, including disconnection, and personal mobile services management. Our original solution, presented in this paper, is a new cloud service called the Cloud Personal

Assistant (CPA) that works on behalf of its user for providing the required cloud services, if they exist – every subscribed user will be allocated an instance of a CPA. With CPA, the responsibility for discovering an appropriate service, invoking it, and getting the result has moved from a client application into the cloud itself. Once the task information has been sent to the cloud assistant from the mobile client, any interruption that occurs on the mobile device has no bearing on the outcome of the service for the task. If the signal is lost, or the battery dies, the task information, invocation and result data are safely in the cloud.

This paper starts with a review of similar work in Section 2. Then, the concept and architecture of the CPA is presented in Section 3. Section 4 gives experimental results, and Section 5 concludes this paper.

II. RELATED WORK

Various approaches have been taken to utilize the benefits cloud infrastructure can offer to mobile devices.

Cloudlets are a form of computing infrastructure located near the mobile device proposed by Satyanarayanan et al [5]. The basis for cloudlets is the need to run applications on mobile devices that operate under near real-time constraints; latency must therefore be minimal. Cloudlet infrastructure is self-managing and can be deployed in public area in a secure enclosure. Generally, a cloudlet will only be one Wi-Fi hop away from the mobile device. The cloudlet runs a minimal virtual machine for operating systems, which can be combined with a virtual machine overlay located on a user's mobile device, containing a user's custom applications and settings. The cloudlet can then carry out some task work for the mobile device user, while sending other work to the cloud. Our approach by having the personal assistant discover and use services asynchronously will remove the need for users to obtain and maintain applications themselves on virtual machines. In addition, the services can be optimised to take advantage of awareness of user context from the mobile device resources such as sensors, which VM's such as Windows 7 are not designed to use.

Code partition and offload is another explored topic. Cuervo et al [7] developed a system called MAUI, which can offload execution of parts of an application, onto cloud infrastructure. Developers annotate suitable application methods as Remoteable, which are then considered as candidates for offloading at runtime. Chun et al [8] implemented a very similar system to MAUI, known as CloneCloud. It functions in the same way as MAUI, by offloading execution to the cloud. Here, a clone of the device runs in the cloud. This gives the advantage that a method offloaded to the cloud can call native methods, even though native methods themselves, like MAUI, cannot be offloaded. CloneCloud does not require the developer to annotate or modify their application in any way. Both solutions profile applications against the current network state, the energy consumption characteristics of the device and application, required resources, and in the case of CloneCloud, time required for execution. In both cases an optimization solver decides at runtime should a method be sent to the cloud if cloud based execution will minimize the objective function. Our approach will save energy and time overhead by not requiring any partitioning, offloading of application code or profiling, as we are using cloud based services. The assistant needs only be signaled to carry out work, possibly using history data, so no consideration of transferring large amounts of data such as application code and data in the above approaches is required. Our approach will also store service results in the cloud until the device is ready to receive them, unlike the above approaches where cloud execution state is lost if disconnection occurs.

Some approaches have attempted to relieve the need to use WAN or remote cloud infrastructure, and instead use information from other devices in close proximity. These devices can create an ad-hoc mobile cloud. One such approach is by Huerta-Canepa and Lee [9], which they call a virtual cloud provider. The idea behind this approach is that users who share the same environment may be interested in performing the same tasks. The example used is two mobile users attempting to translate a Korean text description of a museum piece. Due to roaming charges, the user does not want to access cloud services, but the other mobile user already has this information on the device, so an ad-hoc network is created between the devices to obtain the information. The implementation involves interception of application calls to cloud infrastructure, and modifying it to use its virtual cloud provider. It also determines what devices in proximity are stable (not moving away out of the vicinity), and resource availability to determine if a task needs to be offloaded to another device. Using a cloud assistant approach, the mobile user would not need to spend time and roaming usage searching for a service to translate and getting a response, normally through a web browser with all the other unnecessary larger elements on web pages such as images. The cloud assistant, when given this task, works asynchronously, not requiring a continuous costly connection to the device, and can simply send back the required information with no overhead when the task is complete.

The partitioning of mobile applications into components and distributing them to other nearby devices is another explored topic. Alfredo is a system by Giurgiu et al [1] which partitions applications into components and distributes them among the nearby devices, in various configurations to minimize or maximize an objective cost function, such as energy cost, or throughput. They conclude that computation intensive components could be distributed, leaving only UI components on the device, where the role of the device becomes a viewer. This is similar to a thin client remote display approach by Simoens et al [12] who study the role of mobile devices as thin client viewers to remote applications. A modification to the cloudlet approach by Verbelen et al [6] is similar to Alfredo in that it modifies the cloudlet concept to be a collection of devices in the proximity. Their system also partitions applications into components and distributes them among the devices. Our approach again removes any profiling overhead, and applications do not have to be created or modified to a distributable component design. As already outlined with respect to cloudlets, our design can benefit from the resources on the device for context awareness use with cloud services, whereas with a thin client viewer approach, the device is just used to take UI input from the user. If the remote application is running on a VM with an OS like Windows 7, the thin client viewer approach also suffers from the same drawback as cloudlets as the OS is not optimised for the capabilities of mobile devices.

III. THE PERSONAL ASSISTANT MODEL

A. The Cloud Personal Assistant Concept

The main concept we propose for the mobile cloud is that of a personal assistant that works in the cloud on behalf of its owner. The main benefit is that CPA is always running in the cloud, receives from its owner the set of tasks to execute, discovers the necessary cloud services, invokes them and then delivers the results. There is no need for a permanent connection between the mobile and the cloud, and the results can be delivered when the mobile user needs them.

When a mobile user subscribes to the system, the CPA management service creates a CPA instance and assigns it together with other cloud resources (storage, CPU) to that user – see figure 1.

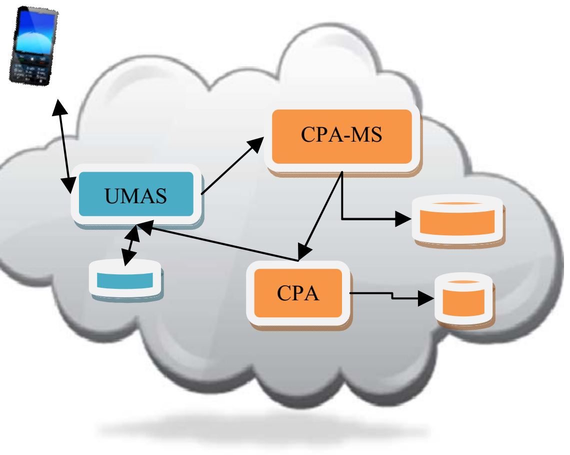

Fig 1. The mobile client subscribes and has allocated an instance of a Cloud Personal Assistant. UMAS - the cloud User Management/Authentication service; CPA-MS – the management of CPA service.

An authentication key is returned to the user as well but the security aspects are not discussed in this paper.

The newly created CPA instance will persist in the cloud as long as the user is subscribed, being either active or dormant.

CPA is given information by the user on some task he/she is interested in carrying out – see figure 2. This information can be the type of service they are looking for, and some parameters related to the task. The information can be given to the cloud assistant from a client application running on the mobile device. The cloud assistant, which “lives” in the cloud, will take this information, search for and discover a service in the cloud which can carry out this task for the user. Once a service has been found, the cloud assistant will invoke the service, passing it the parameters that the user has provided. The cloud assistant will then wait for a result from the service. Once the result has been handed back to the cloud assistant, it is stored, and the user is then notified that the result is ready. The result of the service invocation can be viewed, used immediately or at some moment later in time.

One benefit of this model is that once the user has given a task description to the cloud assistant, even if the battery dies or the signal is lost, the task execution and the details associated with it are safe, as the responsibility for the discovery and invocation of the services is in the cloud, and no longer with the user. This approach also has added benefits. Computationally expensive, intense, long-running tasks, free the user’s client software from having to be left on and running in an uninterrupted state on mobile devices or desktop PCs. It can take advantage of the

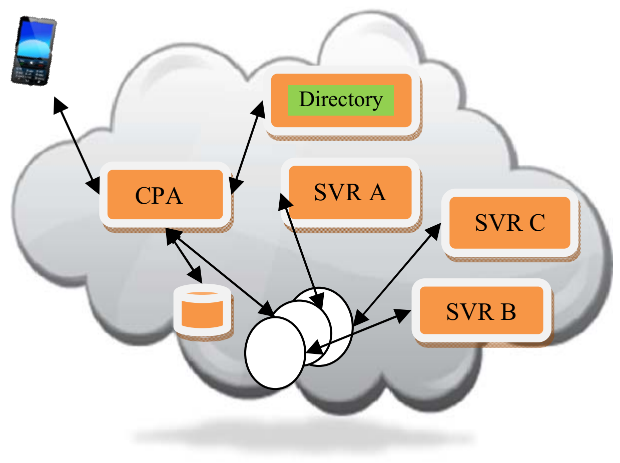

Fig 2. The Cloud Personal Assistant looks up the Directory and then invokes services A, B and C.

excessive resources of the cloud that may be lacking on the mobile device. The user does not need to search for, download, and install software or service client applications on their device.

The difficulties of this approach include the heterogeneity of software and services available, and selecting the right one for the task based on the users inputs. Different services take in and return different types of data. Depending on the user's location and the quality of the network they are connected to, the latency involved in contacting the cloud assistant must be taken into account. Ideally, the cloud assistant and its data should be as close as possible in the cloud to an access point near the user's location.

B. System Architecture

The mobile cloud middleware based on the concept of CPA is divided up into three tiers, with the original intent of deploying the application for each tier onto a cloud based instance – the user tier, the task tier, and the service tier.

The main benefit to this approach is that the system is modular, in that changes to one should not affect the other. This approach also promotes loose coupling, which is very important in large software systems. Database storage is used, to keep data persistent.

The user tier is responsible for user registration, login/logout, and presenting the cloud assistant and related tasks to the user. When a user registers, an entry is made for them in a User table in the database, storing all their details. Upon registration, a cloud assistant instance is created for the user. This is stored in the cloud assistant table in the database. The cloud assistant maintains a reference to its owning user, and lists of current tasks in process, and previous tasks, known as history. The Tasks table contains the tasks and pointers to their parent cloud assistant. Users can create new tasks to be added to the cloud assistant's current task list for execution. A user can log in at any time to check has a task finished execution. The user can also view all previous executed tasks. Any new task the user creates is passed to the task tier.

Tasks are passed into *the task tier* from the front end user tier over a queue. A new task is stored in the Tasks table in the database for history. When a task is passed in, the task handler class will look up the registry for the appropriate service. When one is found, it will create a service access client. This client will then be responsible for contacting the required service, passing it the information, and waiting for completion and results to be handed back from the service.

When the client has the result, it will pass this result back to the task handler, which will update the task as complete in the database. The result may also have to be stored, or it could be passed back over a queue to the cloud assistant.

The services tier is just an abstraction of a container containing the cloud services. Services here will need to register themselves with the registry, and receive tasks from task tier. To handle load and for scaling purposes, they may need to create separate threads of execution for each task a service receives. A thread pool could be utilized here if required.

Finally, the system uses a *discovery service* that allows clients to discover appropriate cloud services.

C. System Implementation

Three cloud providers were evaluated, Amazon Web Services, Microsoft Azure, and Google AppEngine. Amazon was the selected choice as it provides support for running the Tomcat servlet container to deploy Java web-based enterprise applications, by uploading the WAR files to the container. It also supports MySQL databases. To contrast, Azure does not readily provide Java support "out-of-the-box", in that it does not readily run a Java Virtual Machine (JVM) or Tomcat (they must be packaged and deployed with the application). It uses Microsoft SQL Server. The Google AppEngine only recently provided support for Java and it does not use a MySQL based database solution, rather a NoSQL datastore.

The following Amazon Web Services features were used:

- EC2 instances running application tiers;
- RDS which provided MySQL based databases;
- SQS which provided the queue in the first design;
- CloudFront which provided a Load Balancer;
- ElasticBeanstalk which provided automatic application management.

The Task Handler

The task handler is a class which when handed a task description by the cloud assistant, will look up a service in the registry, and create a service client for executing the task. In our first implementation a queue was used to pass task data between the cloud assistant and the task handler on the task tier. Due to queue payload restrictions and time overhead, our second and final implementation removed the queue, and developed the task tier as a component of the cloud assistant, so they could communicate directly. The cloud assistant creates a new thread of execution for a new instance of the task handler for each new task received, so several tasks can run concurrently.

When handed a task, the task handler examines the task to check if a WSDL file URL is already associated with that task. If there is, then the task or a similar task has been executed before, so the entire registry lookup process is skipped for performance. If no previous WSDL file URL is associated with the task, then the task handler uses a library of Apache Scout code, to lookup the jUDDI registry, with the task type information associated with the task. The library encapsulates SOAP level communication over HTTP. The search will return a list of services that have a service name that approximately matches the task type entered. The first result is chosen as the service. In future versions, service attribute negotiation should be considered, but this is beyond the scope of the jUDDI service result implementation.

The task handler extracts the WSDL file URL from the service result and stores it with the task. The task handler will now create a Service Client, which will be responsible for invoking the service and fetching a result. Once the result has been passed back from the Service Client, the task handler forwards it back to the Cloud Assistant for storage and user notification.

If no service was found in the registry that is similar in name to the provided service type of the task, a notification message indicating no service found is sent back to the Cloud Assistant.

The Service Client

The service client is responsible for creating a dynamic client for the remote service, given the WSDL file for the remote service from the task handler. It chooses the relevant operation exposed by the service that matches the operation associated with the task the user submitted. A limitation of the framework is that operations cannot be searched. If no operation is found in the dynamic client created from the WSDL, a notification message is returned to the task handler to indicate no service was found. The name must explicitly match what the user provided as the operation name. It will then wrap up the users input parameters and invoke the service with those parameters. When the result is returned, it is passed back to the task handler.

IV. EXPERIMENTAL RESULTS

The proof-of-concept experiments were carried out on an Amazon EC2 compute instance, running Ubuntu Linux 11.10. The instance size is t1.micro, which includes 613MB RAM, and 2 EC2 Compute Units [10-11].

The following time parameters detail the performance measurements taken in testing, measured in time (seconds) to execute the operations described. The testing results were gathered using timing logging statements placed at important points in the code.

T_d = Time for service discovery. In this test, this involves checking if the task handed to the task handler already has a WSDL URL associated with it, in which case it is a re-run of a previous task. If a former WSDL URL is not found, the jUDDI registry is then queried, and a list of found services returned. The list of services returned is iterated over to pick the service. Only one service is returned from the registry.

T_p = Time for preparation (dynamic client creation, parameter wrapping)

T_i = Time for service invocation (from the time of method call to result returned).

T_t = Total time for discovery, preparation and service invocation. Note that this measurement was not calculated by simply summing T_d, T_p, and T_i; it was calculated using different timing logging statements to the other time measurements, placed at the start and end of the task handling process. It does not take into account the time taken for new task processing on the user tier, e.g. creating new task objects and saving them with the cloud assistant in the database, authenticating the user, and sending confirmation responses to the client.

The first set of experiments involved a cloud arithmetic service that is invoked for the first time. The results are shown in table 1. The large difference between the results of the first run and subsequent runs may be explained as follows: before these tests were run, the server instance was restarted. After the restart, on the first run, the WSDL file was fetched, and the dynamic client was compiled, with its compiled class files stored in a temporary directory. It is possible after the first run, the WSDL file may have been cached, and the dynamic client files were not deleted from the temporary directory as the server was not restarted, and therefore may not have been recompiled.

The second test is similar to the previous test except that it was re-run from a previous task. The results are shown in table 2. Therefore the step of querying the registry and fetching the WSDL file are skipped, as the previous task which the new tasks are run from already have a WSDL file URL associated with it. Therefore T_d is not measured, although the check if the task already has a WSDL file URL associated with it will still take place, and return true. For consistency, the server instance is again restarted before the tests are run. Again, having restarted the instance server before the tests resulted in a longer amount of time required for the dynamic client class compilation in the first run. The time reduction for invoking services already discovered is clearly lower than the time taken when a new service must be discovered. The idea in this is that users will re-use services they have already discovered far more often than searching for new ones each time they need to re-run a task or carry out a similar task.

The time taken to communicate with the cloud assistant from the mobile device was measured. This measurement can vary greatly because the more task data sent (e.g. the sending of a new task and its description data versus simply signaling to the cloud assistant to re-run a task - simply sending the task Id to the cloud assistant), the longer the communication may take.

Table 1. Task: arithmetic Service, no previous service known (e.g. New Task created on client)

Run Number	T_d (s)	T_p (s)	T_i (s)	T_t (s)
Run 1	2.79	2.59	0.001	5.38
Run 2	0.371	0.195	0.001	0.568
Run 3	0.392	0.198	0.001	0.59
Run 4	0.359	0.184	0.001	0.543
Run 5	0.403	0.179	0.001	0.584
Average	0.863	0.669	0.001	1.533

Table 2. Task: arithmetic Service, previous service known (e.g. Re-run of previous task from client, no new parameters specified)

Run Number	T_p (s)	T_i (s)	T_t (s)
Run 1	2.397	0.001	2.399
Run 2	0.193	0.001	0.194
Run 3	0.19	0.001	0.19
Run 4	0.189	0.001	0.191
Run 5	0.186	0.001	0.187
Average	0.631	0.001	0.6322

The client mobile device used for testing is a Samsung Galaxy S2, running the Google Android OS version 2.3.4 (Gingerbread). The network provider is Vodafone Ireland. Two tests took place, communication over the HSPA+ connection on the device, and over the Wi-Fi network connection to Eircom (ISP) Broadband, with a measured download rate of 2272 kbps, and a measured upload rate of 512 kbps. The router providing the modem and Wi-Fi connection is a Netgear N300 Wireless Dual Band ADSL2+ Modem Router (model DGND3300v2) using wireless mode G. To take the measurement, timing logging code was inserted just before and after the HTTP connection is made with the request to the cloud assistant application and the response being received. It does not take into account any of the service work in the application such as JSON parsing, building the HTTP Request, and the sending of user entered data from the activities to the service. From the user tier perspective on the cloud application, this test shows the time taken for what was not measured in the service discovery and invocation tests, namely the user tier processing of new tasks, associating them with the cloud assistant, insertion into the database and authentication of the user. The response is sent to the client only after these tasks processes have been completed. Similar to the previous tests, the time was measured for sending new tasks and re-run tasks to the cloud assistant over the Wi-Fi and 3G networks from the mobile device. The results are shown in tables 3-6. An important factor on timing is the location of the server and the client.

The client device was located in Cork, Ireland. The cloud application running on Amazon's AWS instance servers, and the RDS database servers, were located in the Amazon US-East data centre, located in Northern Virginia, USA.

Table 3. Time taken to send new task to cloud assistant from mobile device client application over residential Wi-Fi network and receive confirmation response. Task was a new task, as in table 1.

Run	Time (s)
Run 1	1.233
Run 2	1.266
Run 3	1.149
Run 4	1.248
Run 5	1.155
Average	1.210

Table 4. Time taken to send a re-run task to cloud assistant from mobile device client application over residential Wi-Fi network and receive confirmation response. Task was a previous task, as in table 2, re-run from client, no new parameters specified.

Run	Time (s)
Run 1	1.178
Run 2	1.09
Run 3	1.116
Run 4	0.981
Run 5	1.108
Average	1.095

Table 5. Time taken to send new task to cloud assistant from mobile device client application over network provider HSPA+ connection and receive confirmation response. Task was a new task, as in table 1.

Run	Time (s)
Run 1	2.116
Run 2	1.976
Run 3	1.069
Run 4	1.056
Run 5	1.465
Average	1.536

Table 6. Time taken to send a re-run task to cloud assistant from mobile device client application network provider HSPA+ connection and receive confirmation response. Task was a previous task, as in table 2, re-run from client, no new parameters specified.

Run	Time (s)
Run 1	1.77
Run 2	1.432
Run 3	1.096
Run 4	1.29
Run 5	1.589
Average	1.435

After the arithmetic service task requests were sent to the cloud assistant, the mobile device client application would be closed down. When the cloud assistant obtains the results from the service, it is stored in the task history. An email is sent to the user informing of the completed task. At this point or at a future point in time, the mobile client application can be opened to retrieve the result from the cloud assistant. This shows that execution of the task and saving the result is possible even when the device is disconnected from a network.

V. CONCLUSIONS

In this paper, we introduced the concept of the cloud personal assistant as the main component of a new mobile cloud middleware system. The proposed cloud assistant solution is based on moving the responsibility of discovering and utilizing services into the cloud on user’s behalf, so if any interruption occurred on the mobile device, the progress of executing some task would be safe in the cloud.

An implementation was designed to demonstrate the concept. It consists of three primary tiers, a user tier, a task tier, and a service tier. A fourth tier is the registry of services. The user tier maintained user and task information, and can send and receive data to the user’s client. The cloud assistant “lives” in the cloud. It uses the task tier to lookup services in a registry. It can then invoke services running on the services tier, where running sample services created for this project are deployed and running, including arithmetic, AWS S3 and AWS RDS services. The result returned from the service that was invoked is stored by the cloud assistant, and the user notified of the completed tasks. The user can view the result at any future moment in time.

The performance of the application was measured and found to be relatively quick, never taking more than three seconds at most to complete any process involved in the server and client applications. Under normal circumstances operations never reached a two second duration. The difference in timing over the tested Wi-Fi network and HSDA+ network were negligible.

Several limiting factors are present that will be addressed in the future. The lack of flexibility in searching for services is a restraining factor on the potential of this implementation, such as service and operation names. Services cannot be described in such a way that would allow for negotiation between services and the cloud assistant in which one service of many possible suitable services should be selected. The difficulties of the heterogeneous services environment make it problematic to deal with the multitude of different inputs and outputs from services.

ACKNOWLEDGMENT

The PhD research of Michael J. O'Sullivan is funded by the Embark Initiative of the Irish Research Council.

REFERENCES

[1] I. Giurgiu, O. Riva, D. Juric, I. Krivulev, and G. Alonso, "Calling the cloud: Enabling mobile phones as interfaces to cloud applications", Middleware 2009: 83-102.
[2] K. Farkas, O. Wellnitz, M. Dick, X. Gu, M. Busse, W. Effelsberg, Y. Rebahi, D. Sisalem, D. Grigoras, K. Stefanidis, D. N. Serpanos, "Real-time service provisioning for mobile and wireless networks", *Computer Communication Journal*, March 2006,
[3]ABI Research, "Mobile Cloud Computing Subscribers to Total Nearly One Billion by 2014", http://www.abiresearch.com/press/1484-Mobile+Cloud+Computing+Subscribers+to+Total+Nearly+One+Billion+by+2014.
[4] D. Linthicum, "Where SOA meets Cloud. 3 SOA/Cloud Trends to Watch in 2011", http://www.ebizq.net/blogs/cloudsoa/2010/12/3-soacloud-trends-to-watch-in-2011.php.
[5] M. Satyanarayanan, P. Bahl, R. Caceres, N. Davies. The Case for VM-based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, Volume 8 Issue 4, October 2009, pp 14-23.
[6] T. Verbelen, P. Simoens, F. De Turck, B. Dhoedt. Cloudlets: Bringing the Cloud to the Mobile User. *MCS '12 Proceedings of the third ACM workshop on Mobile cloud computing and services*, pp 29-36.
[7] E. Cuervo, A. Balasubramanian, DK. Cho, A. Wolman, S. Saroiu, R. Chandra, P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. *MobiSys '10: Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, pp 49-62.
[8] BG. Chun, S. Ihm, P. Maniatis, M. Naik, A. Patti. CloneCloud: Elastic Execution between Mobile Device and Cloud. *EuroSys '11 Proceedings of the Sixth Conference on Computer Systems*, pp 301-314.
[9] G. Huerta-Canepa, D. Lee. A Virtual Cloud Computing Provider for Mobile Devices. *MCS '10 Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond*, pp 6:1-6:5.
[10] Amazon Elastic Compute Cloud AWS Documentation – "Instance Families and Types" - http://docs.amazonwebservices.com/AWSEC2/latest/UserGuide/instance-types.html
[11] Amazon Elastic Compute Cloud AWS Documentation – "Micro Instances"- http://docs.amazonwebservices.com/AWSEC2/latest/UserGuide/concepts_micro_instances.html
[12] P. Simoens, F. De Turck, B. Dhoedt, P. Demeester. Remote Display Solutions for Mobile Cloud Computing, Computer, 2011, 44, (8), pp. 46-53