MOCCAA: A Delta-synchronized and Adaptable Mobile Cloud Computing Framework

Harun Baraki, Corvin Schwarzbach, Malte Fax and Kurt Geihs
Distributed Systems Group, University of Kassel, Wilhelmshöher Allee 73, Kassel, Germany

Keywords: Mobile Cloud Computing, Delta Synchronization, Resource Management System.

Abstract: Mobile Cloud Computing (MCC) requires an infrastructure that is merging the capabilities of resource-constrained but mobile and context-aware devices with that of immovable but powerful resources in the cloud. Application execution shall be boosted and battery consumption reduced. However, a solution's practicability is only ensured, if the provided tools, environment and framework themselves are performant too and if developers are able to adopt, extend and apply it easily. In this light, we introduce our comprehensive and extendable framework MOCCAA (MObile Cloud Computing AdaptAble) and demonstrate its effectiveness. Its performance gain is mainly achieved through minimized monitoring efforts for resource consumption prediction, scalable and location-aware resource discovery and management, and, in particular, through our graph-based delta synchronization of local and remote object states. This allows us to reduce synchronization costs significantly and improve quality dimensions such as latency and bandwidth consumption.

1 INTRODUCTION

Companies utilize the computational power of cloud resources, for example, to speed up their processing, to compensate peaks under high loads or to reduce costs by consumption-based billing and simplified maintainability. Any such support by Cloud Computing would also extend the capabilities of resource-constrained mobile devices. While mobile devices have to avoid energy and computational intensive applications, clouds are designed for these particular use cases.

Although at first glance both Mobile Computing and Cloud Computing fit perfectly together, many challenges have to be overcome to trigger a wide and straightforward use of MCC. The core questions in MCC are *what to offload* (which part of a mobile application) and *how, when and where to execute* the resource-intensive part remotely. In the last few years, several approaches have been developed and examined in the research community. Most of them deal with one or two aspects of MCC, but do not consider the downsides and effects for the other dimensions. Approaches like (Chun et al., 2011) and (Yang et al., 2014) relieve the developer by deciding autonomously what to offload, but need synchronized Virtual Machine (VM) images and adapted operating systems. Other concepts like (Kemp et al., 2012) and (Giurgiu et al., 2009) bear on the developer's experience and extensive code adaptations to be able to offload application parts. The approach of Ou et al. (Ou et al., 2006) calculates precisely what to offload where, but does not consider the high monitoring and synchronization costs. Further related works are discussed in detail in section 8. The key point is that a comprehensive solution for MCC must handle all core questions mentioned above simultaneously. Nonetheless, the solution has to be applicable to a wide range of mobile applications and should neither slow down the development of applications considerably nor affect their usage by the end user. The objective of our work is to provide a flexible but also comprehensive and comprehensible solution where each of the following parts can be coordinated easily by the whole framework and where each part contributes to the overall efficiency of the system:

- *DRMI*: An asynchronous Remote Method Invocation that applies our performant delta synchronization to ensure state consistency between client and server.
- *InspectA*: A tool that helps developers to detect resource-intensive methods, to create prediction functions for their resource consumption and to minimize monitoring costs by extracting the most relevant features for prediction.

Baraki, H., Schwarzbach, C., Fax, M. and Geihs, K.
MOCCAA: A Delta-synchronized and Adaptable Mobile Cloud Computing Framework.
DOI: 10.5220/0006701101360147
In *Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018)*, pages 136-147
ISBN: 978-989-758-295-0

- *MOCCAA-RMS*: A self-configuring, scalable and distributed resource management and discovery system.
- *MOCCAA annotations*: Android/Java annotations that enable applications, for instance, to execute methods through DRMI on one or more remote resources found by MOCCAA-RMS whenever a prediction function of InspectA recommends it.

On the one hand, straightforward mechanisms and tools shall allow developers to apply MCC also for small and simple scenarios. However, these mechanisms can be combined to implement more complex scenarios. On the other hand, our most basic mechanism, an asynchronous RMI invocation, shall be as fast and compact as possible, especially for typical MCC payloads, that is, large messages that are processed mutually on client and server side. Examples of use illustrate the course of action and prove the practicability of our solution. In addition, our experiments show a significant reduction of latency and message size compared to the default Java RMI.

The remainder of this paper is organized as follows. Section 2 provides first an overview of our framework and introduces each component briefly. InspectA, MOCCAA-RMS, the MOCCAA annotations and DRMI are explained in Section 3 to 6. Section 7 presents our experiments and their results. Section 8 discusses relevant and related works. Section 9 will conclude this work with a summary and ongoing and future work.

2 FRAMEWORK OVERVIEW

A fundamental requirement in MCC is that applications shall also be executable without a network connection. In general, this implies that the whole application is installed on the mobile device. In case sufficient remote resources are accessible, parts of an application are deployed and executed remotely. The granularity of these parts varies depending on the approach. Our approach works at the level of method and object granularity.

At design time the developer has to decide which method and object can be offloaded. Our tool InspectA will help him to detect objects and methods that are very resource consuming and that do not have cyclic dependencies on their invoking methods. An unfavorable constellation would be, for instance, a method *A.a1()* calling a method *B.b1()* while *B.b1()* itself would call, maybe indirectly, method *A.a1()*. In that case, *B.b1()* should only be executed remotely when *A.a1()* is offloaded too. Section 3 presents InspectA in detail.

Having established the candidate objects and methods, the developer has to annotate them as shown in Listing 1 and 2. The annotation *@InjectOffloadable* is used to mark the concrete object reference. In this way, our framework is injecting a proxy that holds an instance of the class referred to. Dependency Injection is well-known to most developers and does not impose a high burden on them. As demonstrated in Listing 1, the objects' methods can be invoked as usual and do not need any special treatment.

```
public class MyClass {

  @InjectOffloadable
  ComplexCalc complexCalc;

  public int m1(int p1, String p2) {
    complexCalc.setVar1(p2);
    complexCalc.doComplexCalc(p1);
    ...
    System.out.print(complexCalc.
        getVar1());
  }
}
```

Listing 1: Annotating the offloadable object.

The injection is implemented through bytecode manipulation and carried out by means of the Javassist library (Chiba, 1998). The injected proxy intercepts all method invocations on the instantiated object and decides whether a local or a remote execution shall take place. The default behaviour proceeds with a local execution. In contrast, invocations of methods, which are labelled with the *@Offload* annotation, lead to further analyses that evaluate the availability of applicable remote resources given through the *@OffloadServers* annotation.

```
@Offloadable
@OffloadServers({ipAdress1, ipAdress2})
public class ComplexCalc {

  String var1;

  @Offload
  public void doComplexCalc(int p1) {
      ...
  }
  ...
}
```

Listing 2: Annotating the class and the method.

This behaviour can be adapted through a plugin architecture. Adding the annotation *@OffloadEvaluator(...)* to an *@Offload* method, a class implementing our Evaluator interface can be specified. The

Evaluator will then be instantiated by the proxy and used whenever the *@Offload* method is invoked. The method parameters and the proxied object are handed over to the Evaluator so that it is capable to decide with the aid of the current content and size of the parameters and object whether a local or remote execution are favorable.

Assuming that the server part of our framework is installed on the addressed servers, this configuration is already sufficient to employ basic MCC functionality. Whenever a method is called remotely, the actual proxied object will be synchronized with the server side. The first time, the whole object graph will be transferred. After executing the method, the delta will be transferred back and applied to the object graph on client side. A second method invocation on the same object would just transfer the delta from client to server and subsequently call the method. Section 6 explains further details.

Although this configuration provides basic MCC functionality, it evinces a lack of flexibility when it comes to the distribution of workload. Servers might be well utilized, suffer from bad communication links or even be down so that alternative servers have to be employed. Section 4 extends therefore our framework with server monitoring and distributed registries. In addition, many scenarios could benefit from a parallel processing of the workload on one or multiple servers. To support this, we introduce in Section 5 additional annotations that allow developers to split tasks and deploy them on multiple servers and aggregate the results on one server or one device. Flexibility is also demanded when it comes to the consideration of devices with different or varying resource capacities. The decision to offload may depend on the current performance of the mobile device. Section 3, in particular section 3.2, explains how the resource consumption of methods can be estimated in advance and demonstrates with the aid of an Evaluator example its use in this sense.

3 InspectA

3.1 Offline Analysis

The InspectA tool helps developers in analyzing their applications in a fine granular way. The current version works for Java 1.8 and Android SDK versions greater than or equal to 23 (Android 6.0). Starting InspectA on the developer's computer, it will connect with the Android Debug Bridge (adb). This allows him to display running applications and their overall resource consumption within a given time frame. The

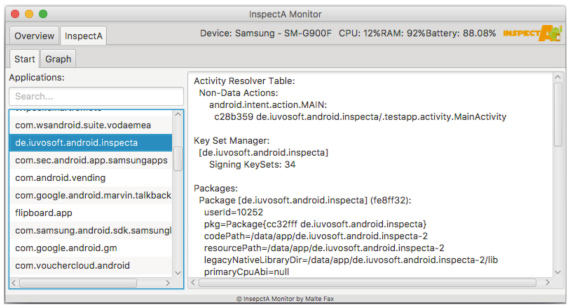

Figure 1: Application information.

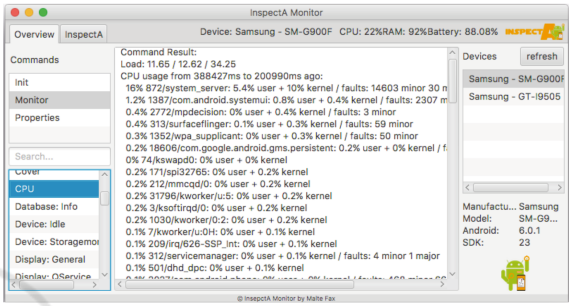

Figure 2: Monitor view listing all running applications and their overall resource consumption in a given time frame.

initial views are depicted in Figure 1 and 2. Information such as CPU and memory consumption can be listed, but also details like obtained and demanded permissions and package specifics.

To obtain more detailed information, the developer has to include our InspectA.aar archive and our Gradle commands into the build process. These start an AspectJ compiler that weaves Aspects around the applications' methods. After each method the current thread's CPU time, the memory usage, the battery status and the execution time are recorded. During the application's execution a dump is created on the mobile device that can be imported into InspectA. Figure 3 shows the detailed view with method granularity. The information can now be sorted in descending order so that methods with high memory, CPU, battery or time consumption can be detected easily. Indeed, Android provides so called tracers for battery, memory and CPU. However, the dumps are created separately for each dimension and the analysis requires much experience.

Sector 4 in Figure 3 displays the invocation path of the monitored methods. All information, including the invocation path and frequency, are stored as a GraphML file that can be further analyzed. Listing 3 presents an excerpt of a GraphML file that is showing a single invocation of *MainActivity.mapGUI* by *MainActivity.onCreate*. Section 3.2 indicates how this file can be utilized to generate prediction functions that

estimate the future resource consumption by means of certain monitoring points. Such functions can be applied in Evaluators as decision-making support.

```
<?xml version="1.0" encoding="UTF-8"
    standalone="no"?>
<graphml xmlns="..." ...>
<graph id="de.iuvosoft.android.
    inspecta" edgedefault="directed">
...
<key id="battery" for="node" attr.name
    ="battery" attr.type="double"/>
<key id="ram" for="node" attr.name="
    ram" attr.type="double"/>
<key id="cpu" for="node" attr.name="
    cpu" attr.type="long"/>
<graph id="0" edgedefault="directed">
<data key="name">de.iuvosoft.android.
    inspecta.testapp.activity.
    MainActivity</data>
<node id="1">
<data key="name">onCreate</data>
<data key="counter">1</data>
<data key="time">347366303</data>
<data key="battery">-1.0</data>
<data key="ram">5.0190162658</data>
<data key="cpu">160813228</data>
</node>
<node id="2">
<data key="name">mapGUI</data>
<data key="counter">1</data>
<data key="time">110729</data>
<data key="battery">-1.0</data>
<data key="ram">0.0</data>
<data key="cpu">102188</data>
</node>
...
</graph>
...
<edge id="4" source="1" target="2"/>
...
</graph>
</graphml>
```

Listing 3: GraphML file generated by InspectA.

The final step at this stage is to annotate the resource-intensive methods and the associated objects. The developer may apply now a check for cyclic invocations between the invoking method and the remote or annotated method. For this purpose, we do not investigate the previously created GraphML dump since it is very likely that it does not cover all possible execution paths. Instead, a System Dependence Graph is generated by means of the static code. Using the JOANA library (Graf et al., 2013), Android as well as Java applications can be analyzed. The JOANA API allows us to query whether the invoking method is reachable by the invoked method. The developer will be informed in case a breach is detected. Adapting and restructuring the code is recommended then.

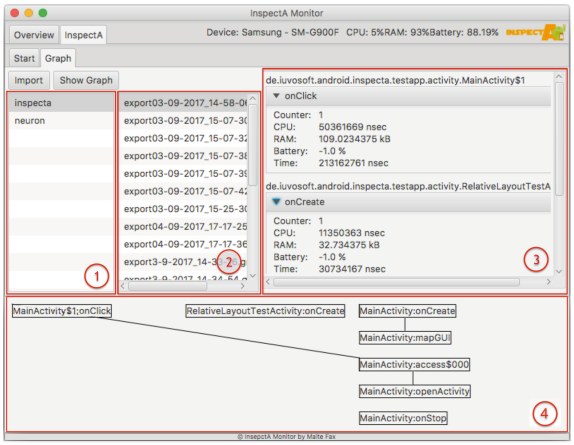

Figure 3: Invocation graph and method details. Battery consumption is set to *-1* in case the mobile device is connected to the power supply.

3.2 Resource Estimation

In a separate work, we develop and examine different prediction techniques for our Evaluators. As the Evaluators are part of the framework and its utilization, we discuss and present the general procedure briefly in this section.

A straightforward method to determine the resource-intensive parts of an application would be to take the average or maximum resource consumption of each method during offline tests and mark those with a high consumption as potential candidates for offloading. Such a pure offline profiling is used, e.g. in (Giurgiu et al., 2009) and (Chun et al., 2011), and was also applied in the previous section. The absence of online monitoring costs is its strength, the lack of flexibility required for different devices and scenarios its drawback. Other approaches ask the developers to mark the resource-intensive parts and the parts that have to be monitored (Cuervo et al., 2010; Kemp et al., 2012) or monitor each aspect of the running application (Ou et al., 2006). However, monitoring quality dimensions like CPU utilization, memory consumption and battery usage requires itself a significant amount of resources and should be avoided during runtime. In addition, operating systems may deny or not provide such information during runtime. This is also true for Android 7 and upper versions which refuse access to such information due to security reasons. The following approach tries to mi-

nimize online monitoring costs and simultaneously to retain a good estimation of the future resource demands of annotated methods.

While InspectA was weaving aspects into the developer's code to monitor, amongst others, the CPU time and memory consumption of each method, now the analysis during design time obtains the size of passed parameters too. In case of primitive data types such as Integer and Double, the value is taken as size. In case of arrays and collections, the number of contained elements is recorded. The developer should then run his application with various input data so that variable execution times are measured. Since testing should be anyway part of the application development, this process step can also be included in JUnit tests.

The order of method invocations is recorded for each thread so that graphs are obtained. The edges of a graph represent the communication between methods, in particular, the frequency of invocations and the parameters and their sizes. Methods are considered as vertices annotated with their consumed time, battery, memory and CPU time.

Subsequently, a Correlation-based Feature Selection (CFS) (Hall, 1999) is applied that is intended for finding relevant monitoring points whose features, particularly parameter sizes and execution times of methods, correlate with the resource consumption of an annotated resource-intensive method that is invoked later in the course of execution. It is possible that a high number of methods exhibits a strong correlation which would correspondingly lead to a high number of monitoring points. However, using CFS, features that correlate with each other, and, hence, are redundant information, are penalized. This leads to a reduced set of potential monitoring points.

Additionally, we identify features that are suitable to predict the resource consumption of multiple annotated methods. Figure 4 depicts a typical execution path. Method 1 is invoking successively methods 2 to 5. The latter invokes methods 6 and 7. Assuming that method 4 and 7 are annotated, method 2 and 3 and the parameters of method 1 (method 1 cannot be considered completely as it is finished after method 4) come into question to predict methods 4 resource consumption. Accordingly, method 2, 3, 4 and 6 and the parameters of method 1 and 5 can be considered for method 7. By combining a Wrapper feature subset selection (Kohavi and John, 1997) with CFS, we reward those monitoring points that can serve as feature for various annotated methods. In the example of Figure 4, method 2 to 3 and the parameters of method 1 are considered preferably as they build the path intersection for method 4 and 7.

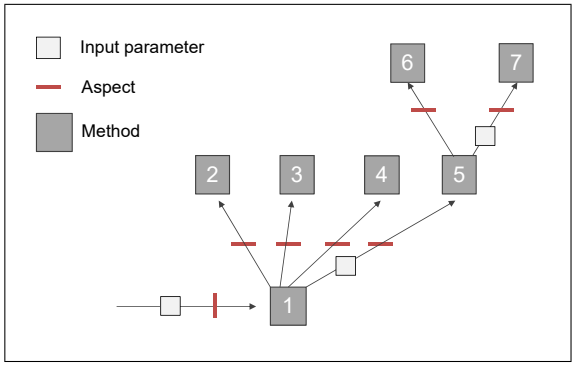

Figure 4: Monitoring method 1, 2 or 3 and their inputs may be sufficient for estimating the resource consumption of following methods 4 and 7.

To further reduce monitoring costs, we restrict the feature selection by involving exclusively the methods' execution times and parameter sizes. Other dimensions like CPU times and memory and battery consumption are ignored. However, during design time we include the latter dimensions for annotated methods. The target is to associate them with execution times and parameter sizes of previously called methods.

Therefore, we apply finally prediction techniques like multivariate adaptive regression splines (Friedman, 1991) and neural networks. Being aware of the relevant features, for each annotated method a model is built at design time. In case of regression splines, the determined basis functions of a regression function will retain their form during runtime. However, since different mobile devices may behave differently, the weights and knots of the basis functions are adapted during runtime if significant discrepancies occur. A detailed description including an evaluation of different prediction techniques is part of a parallel work.

Lastly, an Evaluator can call the prediction functions to receive the estimated resource consumption of a given annotated method. This information cannot only be used for deciding about the offloading step, but also to find suitable servers. The following section provides further details.

4 MOCCAA-RMS

Knowing the resource demands of the client, a matching server has to be searched that can process the user request. Offloading shall not only reduce the energy consumption of the mobile device but should also reduce the overall response time of the application. To achieve that, the following aspects and requi-

rements have to be considered.

A fast resource discovery requires registries that have a low time and message complexity to find suitable servers. What is even more important, is that servers supporting the application's execution are as close as possible to the client. While the resource discovery is just needed at the beginning, an intensive communication may take place between the client and its offloaded part on the discovered server.

A further important requirement is the ease of use for app developers. A reliable Resource Management System that automatically configures and monitors servers and registries and allocates tasks is desirable. Furthermore, it should scale well whenever more resources are added to the system.

Assuming that a developer starts with one server, mostly due to testing or during design time, he has first to install our framework on it. In brief, the framework encompasses firstly a Java application that is loading Java archives and invoking methods on them received through our Delta-synchronized RMI. It is accessible through REST interfaces for application clients (e.g. to search servers and upload jars) and other collaborating servers (e.g. to search for nearby registries and configuration information). Furthermore, it monitors currently running Java applications and the available and used resources of the local system by making use of the SIGAR library [1]. JMS (Java Messaging Service) is used to receive and send aggregated monitoring data from and to other servers. Detailed monitoring data is stored locally in MongoDB[2], a NoSQL database.

If the developer wants to extend the system by adding a further server, he just has to install the same framework and setting the IP address and credentials of the first server in a configuration file. This is also true for any further server so that a VM image can be created and deployed to newly added servers. The following sections explain how the overall system is designed and how it is growing, monitoring and configuring itself automatically.

The first server is also considered as the root of our resource management and discovery system MOCCAA-RMS. The developer can change this by adapting a configuration file. The servers, which may serve as working nodes as well as registries that redirect to other nodes, are structured hierarchically. New servers entering the system are asking through REST the root node for the closest registry. This is determined through an IP to Geo localization. The selected registry may further redirect to child registries by the same procedure. The finally chosen registry can reject if it cannot handle more servers due to bandwidth, processing power or other constraints. In that case, depending on the capacity of the upper registries, a new level or a new sibling registry will be created. For this purpose, a clustering will rearrange the assignment of nodes to their registries, taking into account the newly-added server as a registry, and adapting the registries' databases. Our current implementation is therefore using the k-medoids clustering algorithm (Kaufman and Rousseeuw, 2009).

[1]http://support.hyperic.com/display/SIGAR/Home

[2]https://www.mongodb.com

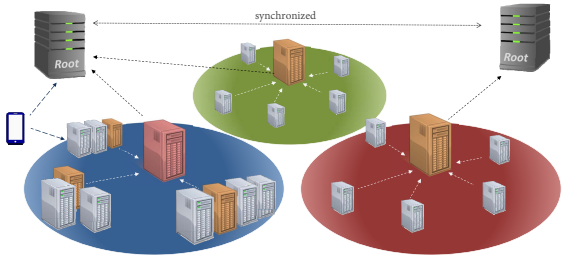

Figure 5: A user smartphone making use of the hierarchy of registries. Orange nodes represent leaf registries, grey nodes servers that are monitored.

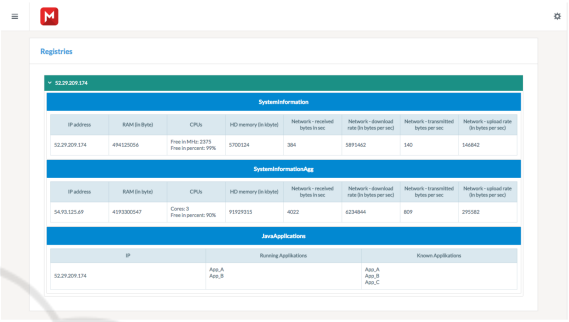

Figure 6: Registry (System Information) with one child (System Information Aggregated).

Registries have to know about the free resources of their subsequent nodes to be able to assign a user request to a server. However, monitoring all servers periodically and reporting to registries could overstrain bandwidth and upper level nodes. Message and time complexity would rise considerably with a growing number of servers. In case of MOCCAA-RMS,

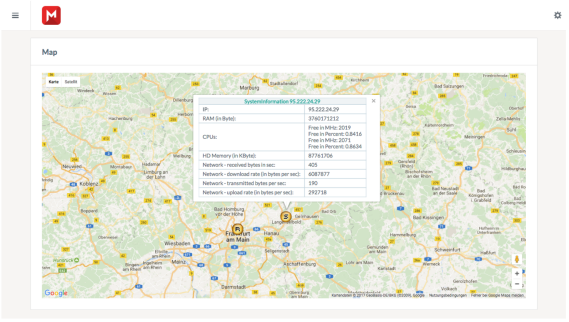

Figure 7: Server details on a map.

only the second last nodes, i.e. the parents of the leaf nodes, are informed about the status of the servers. These registries are highlighted in orange in Figure 5. As soon as a leaf node is assigned to a user request, the orange parent node is subtracting the estimated resource consumption from the servers current status vector available in the registries database (e.g. memory, CPU). The resulting vector stands for the remaining free resources on the server. If the requirements profile is changing during the interaction of the client with the server or when the client finished, the server will report the modification to the parent node. Nonetheless, servers also send through JMS every minute a heartbeat message and aggregated monitoring data to their parent nodes to inform them about their reachability and to correct estimated values.

The parent registries (the red node in Figure 5) of the second last nodes (orange nodes) and upper level registries (black nodes in Figure 5) only get an aggregated summary (vector) of the free resources of their child nodes. In the example of Figure 5 that would be the aggregated values of the three orange nodes that serve as registries for the grey nodes. Updates are only sent when major changes with respect to the last report occur. This avoids a higher number of messages at higher levels of the hierarchy and allows load balancing.

Due to reliability reasons, sibling nodes know each others IP addresses and the grandparents one automatically through their parent node. Whenever a parent node is failing or removed, child nodes elect a new parent node.

A client is usually querying the last contacted local registry, if he did not move more than a configured distance and if it is not the first request, otherwise a close registry has to be found through upper level registries first. The chain of registries between root node and leaf node are also reported to the client. Listing 4 shows the parameters that are sent by the client to a registry's REST interface. If a local registry does not find any suitable servers, the request is redirected to the upper registry. This registry knows roughly about the situation of his child registries and their succeeding servers. It can be thought of as a graph whose edges are annotated with flow capacities. The request will then be redirected to a registry with servers that are underutilized.

Instead of listing potential servers with *@OffloadServers*, the developer has to indicate the root registry's address through *@OffloadRegistries* in the application. In case redundant root registries are available, all of them can be implied. Our framework offers additionally a Java Web Archive that can be deployed on an application server. It accesses the root node and traverses step by step the child registries. It creates an overview for the developer or an administrator and informs them about servers that are down or not accessible anymore. Figure 6 shows a root node with one child node. A map view is provided too (Figure 7). It should be noted, that a registry is considering its own free resources as available for clients too.

```
@GET
@Path("server_search")
@Produces(MediaType.APPLICATION_JSON)
public ServerInfo getServer (
@QueryParam("app_name") String
    app_name,
@QueryParam("app_version") String
    app_version,
@QueryParam("latitude") double lat,
@QueryParam("longitude") double lon,
@QueryParam("ram") long ram,
@QueryParam("cpu") long cpu,
@QueryParam("message_size") long
    message_size) {...}
```

Listing 4: Method signature of the registries search method.

MOCCAA-RMS is tailored to use cases where the consumption can be roughly estimated. This allows an efficient monitoring and discovery of servers. A configuration by the developer or administrator is only needed when default values and thresholds shall be changed.

5 PARALLEL EXECUTION

While a single *@Offload* annotation supports the remote execution of a method on a server, a chain of *@Offload* methods allows the server to offload the received object again and execute the next method on another server.

Listing 5 shows two methods *doComplexCalc* and *doComplexParallelCalc*. While *doComplexCalc* is executed on the server with the address *ipAddress1*, it delegates the execution of *doComplexParallelCalc* to another server. The annotation ***@Splittable*** effects that the array is splitted into multiple arrays. Hence, the method *doComplexParallelCalc* is invoked on the servers with the addresses *ipAddress2, ipAddress3, ipAddress4*, but each of them receiving another part of the array. Each of them could store its results in another field or another part of an array. The implementation makes use of the OpenHFT Chronicle Engine [3] to distribute the tasks. Our annotation is experimental

[3]https://github.com/OpenHFT/Chronicle-Engine

as there is no check for side effects. The developer should be aware of the Bernstein conditions listed in (Bernstein, 1966) and, in specific cases, further conditions mentioned in (Chaumette et al., 2002). In future, further patterns will be worked out. Additionally, the JOANA library shall be involved to support developers in detecting side effects.

```
@Offloadable
public class ComplexCalc {

 String Name;
 int[] resultArray;

@Offload
@OffloadServers(ipAddress1)
 public void doComplexCalc(int p1,
         URL[] imgURLs) {
    ...
    doComplexParallelCalc(imgURLs, p2)
        ;
    ...
 }

@Offload
@OffloadServers({ipAddress2,
  ipAddress3, ipAddress4 })
 public void doComplexParallelCalc(
    @Splittable URL[] imgURLs, int p2)
    {
    ...
 }
}
```

Listing 5: Chain of @Offload and @Splittable.

6 DRMI

In the scope of this work, the problem of synchronizing object graphs between clients and servers is addressed too. The related work known so far transfers the whole state or, at least, the state that is addressed on client and server side. Our goal is to communicate only the differences between the object states located at the client and the server. That means, for instance, that after the client has sent the required state to the server, the server will subsequently return the delta between the new and the old state after executing a method on it. In a following request, the client may send itself just the delta to the server.

The first step is to extend the relevant classes with an identification field *offloadId* that enables us to assign objects unique IDs during runtime. Again, this step is done through bytecode manipulation. Unique IDs are essential since objects may, for instance, be deleted while at the same place a new, similar one can be created. Figure 8 illustrates this scenario. The white object graph is transferred to a server. The invoked method replaces object 3 by a new object 4. Synchronizing back the object graph, the algorithm has to detect whether to replace or to add object 4. Algorithms like rsync or xdelta (Tridgell, 1999) that work on streams and are thought for file systems cannot detect such deltas.

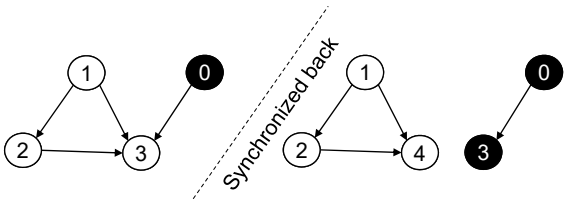

Figure 8: Object 3 replaced on server, but still referenced by object 0 at client side.

During runtime, the proxy intercepts calls on *@Offload* methods. If the Evaluator decides to execute remotely, the proxy serializes initially the object graph by first numbering the unnumbered objects and applying then the Kryo serialization library [4]. The benefit of Kryo is that classes do not need to implement Java's Serializable interface and that it works on Android too. Furthermore, it compresses the serialized file more efficiently (Zhao et al., 2016).

The method parameters are also serialized and transferred together with the object graph and the method name. However, parameters are passed by value, the object and its member variables behave like passed-by-reference. In this way, the developer can decide what shall be synchronized back and what not.

After the remote method invocation, the proxy returns immediately since we apply an asynchronous RMI. The client application can continue its execution as long as it is not invoking any further methods on the object. In the latter case, for example after invoking a getter on the object, the proxy would block until the object is synchronized back. In case a timeout is reached or the communication link is broken, our proxy invokes the local method.

We decided on an asynchronous RMI as we expect high workloads and long execution times. Nonetheless, the developer should take into account not to access any objects from the transferred object graph as long as it is not synchronized back. For this purpose, he should invoke a method such as a getter on the object so that the proxy blocks until synchronization is finished, or access the object graph in general through methods of the annotated object as it is proxied.

[4] https://github.com/EsotericSoftware/kryo

Table 1: Experiment 1 - Locally executed RMI.

	Message sizes in Bytes				Roundtrip time in ms	
Number of objects	**Submitted (DRMI)**	**Received (DRMI)**	**Submitted (Java RMI)**	**Received (Java RMI)**	**DRMI**	**Java RMI**
10	78	35	241	241	5.0	2.5
100	708	35	1411	1411	6.2	4.4
1000	7009	35	13111	13111	7.0	21.4
10000	70009	35	130111	130111	30.4	62.3
20000	140010	35	260111	260111	35.2	72.3
30000	210010	35	390111	390111	52.9	343.0

On server side, the framework deserializes the object graph and parameters and invokes through Java Reflection the method with the given parameters. A copy of the received original graph is kept in memory. After method execution a depth-first search is comparing the changed and the original graph and noting down all differences by recording the concerned object ID, the field ID (field position), the new value (ID in case of reference), and a code that is describing the required command, for example, resizing an array, assigning a value, adding or removing an element from a collection. Due to unique IDs cyclic references do not pose a problem during the graph comparison. Newly added objects are assigned IDs from a new range of successive IDs that does not intersect with that of the client. The recorded deltas and the compressed and new objects are transferred back by invoking a callback function on the client. On client side, the commands are applied through reflection.

Table 2: Results of experiment 2 with a bandwidth of 10 MBit/s for upload and 42 MBit/s for download.

Number of objects in object graph	**Roundtrip time in ms**	
	DRMI	**Java RMI**
10	35.5	32.4
100	36.7	34.9
1000	39.3	56.0
10000	65.4	116.1
20000	71.9	154.8
30000	84.2	448.4

On client side, the previous serialization file is replaced by the serialization of the updated and synchronized version of the object graph. This allows the client to transfer himself only the deltas to the server when invoking a remote method on the object a second time.

The following section compares Java RMI and our Delta-synchronized RMI (DRMI) and discusses their advantages and disadvantages.

7 EVALUATION

The first experiment is carried out on a local computer. Both the client and the server are running on the same device, a notebook with 8 GB of memory, an Intel i5 processor with two 2.6 GHz cores and Java 1.8. All used libraries, including Kryo, Javassist, AspectJ and our proxies and Evaluators can run on Java 1.8 as well as on Android 6 and upper. This allows an application for Cloud Computing as well as Mobile Cloud Computing. However, since we compared our approach with Java RMI, which is not suited for Android, we restricted the following experiments to Java VMs.

For the experiment, a class *Node* is implemented. It has one Integer field and two Node references so that a tree was created. On the root node, we invoked a method with an Integer parameter that changed the Integer field of a random successor node. The method was executed on the server which required first the whole object graph. In case of DRMI, the delta was returned then after method execution and applied to the client's object graph. In case of Java RMI, the whole object graph had to be returned to stay synchronized. Each value in Table 1 shows the average of 100 runs.

The local execution shows the time overhead independent of the used network and bandwidth. The object graphs with 10 and 100 nodes are processed and synchronized faster with Java RMI. DRMI has first to create copies of the received object graphs and traverses the object graph to detect the changes. At a certain level, copying and traversing the object graph with DRMI is faster then serializing with Java RMI. The test with 1000 nodes shows that DRMI performs faster than Java RMI when the graph gets more complex and the changes are small. Since only one Integer is changed, the serialization costs for the delta are negligible then.

Regarding the message size, the response of DRMI retains its size as expected since all test scenarios change one Integer value. Hence, the new Inte-

Table 3: Experiment 3 - Locally executed. 10% of all Integers are updated.

		Message sizes in Bytes				Roundtrip time in ms	
Number of objects	**Number of deltas**	**Submitted (DRMI)**	**Received (DRMI)**	**Submitted (Java RMI)**	**Received (Java RMI)**	**DRMI**	**Java RMI**
10	1	78	35	241	241	5.3	2.4
100	10	708	116	1411	1411	6.1	4.3
1000	100	7009	963	13111	13111	8.3	21.8
10000	1000	70009	9964	130111	130111	37.0	63.5
20000	2000	140010	19964	260111	260111	46.7	74.2
30000	3000	210010	29964	390111	390111	61.3	339.3

ger value, an assign command, the field position and the object ID and header information are returned. Compared with Java RMI, the request message size is smaller too due to a compression with Kryo.

Repeating the same experiment over a network with a client upload rate of 10 MBit/s and a download rate of 42 MBit/s, we obtain the results listed in Table 2. The message size stays the same like in experiment 1. The average ping time between client and server were 30 ms. Compared to Java RMI, DRMI has a lower increase in roundtrip times due to small message sizes. Restricting the bandwidth further would be in favour of DRMI.

Experiment 3 complies with the conditions of experiment 1 except that 10% of all Integer values are changed. Thus, the number of deltas is increased. Table 3 shows that the response message size grows and that it almost reaches $1/10$ of the Java RMI message sizes. Each delta requires around 10 Bytes. As expected, the roundtrip time also increases slightly. However, the performance does not decrease much.

The experiments lead to the conclusion that an option should be provided for developers, if they want to deactivate delta calculations and always transfer the whole object graph. This is recommended whenever message sizes are a few bytes to kilobytes or whenever most elements of the object graph are changed. In contrast, determining and transferring deltas is highly suitable when few modifications are expected and message sizes are a few kilobytes, megabytes or bigger. In addition, it can also be preferred when clients often invoke methods on the same object.

8 RELATED WORK

The different proposed methods in the area of MCC can be distinguished firstly with respect to the granularity and structure of the partitions. Their design affects largely the flexibility and efficiency of the offloading mechanism and the layout of the corresponding execution environment of the cloud counterpart. Furthermore, it may also have an impact on the requirements the developer has to adhere when developing the mobile application. In this view, we will discuss frequently cited related works in the following and focus particularly on their partitioning approach, their synchronization mechanism, and the overhead for developers and users.

Chun et al. (Chun et al., 2011) employ device clones running as applications-layer virtual machines to enable an offloading approach that does not require any preparatory work by software developers. Hence, their CloneCloud framework supports unmodified mobile applications as well. A static analysis tool discovers first possible migration points with respect to a method- and thread-level granularity. This implies that methods accessing certain features of a machine or methods that share their native state with other methods, have to stick together. A dynamic profiler captures then the execution and migration costs for randomly chosen input data and creates a profile tree that depicts the method invocations and their costs, e.g. their execution time. Finally, an optimizer selects the migration points that reduce the total execution time or the energy consumption of the mobile device by considering the computation and migration costs. During runtime the virtual state, the program counter, registers, heap objects, and the stack will be offloaded as soon as methods marked with migration points are invoked. The downside of this approach is that it demands for a synchronized clone. Consequently, the authors presume in their evaluation that a VM is instantiated already and that data, applications and configurations were available in the VM. They do not cover the additional time for the instantiation and synchronization steps and the costs of maintenance, bandwidth and leasing of the VM.

In (Cuervo et al., 2010) Cuervo et al. introduce MAUI, a system that is similar to the aforementioned framework CloneCloud. Like CloneCloud, MAUI operates at the level of method granularity and requires a VM as device clone. However, MAUI is asking the developer to annotate the methods that shall

be considered for offloading. During runtime, MAUI checks the resource consumption of these methods by serializing the required member and static variables and by monitoring the CPU cycles and the execution times. Using the two latter values, the energy consumption can be estimated with a linear regression model. By incorporating additionally the bandwidth, the latency, and the size of serialized data, MAUI is able to evaluate the costs for offloading the code and migrating the states and to decide finally where the method will be executed. In contrast to CloneCloud, it offers an online profiling, however, to the detriment of a higher resource consumption. And due to the VM synchronization, MAUI is exposed to the same handicaps like CloneCloud.

ThinkAir (Kosta et al., 2012) from Kosta et al. is a MCC framework that was derived from CloneCloud and MAUI. Developers shall only need the *@Remote* annotation for their method to execute it remotely. However, further details for developers are not explained. Functions for resource consumption prediction for previously unknown methods are learned online, that is, during runtime. Learning and capturing such monitoring data requires many resources and usually slows down the application. Information like CPU time, display brightness, Garbage Collector invocations and many more are provisioned. In addition, six different Virtual Machine (VMs) types are provided. A developer may parallelize his methods and execute them in parallel on these VMs.

Techniques like cloudlets (Satyanarayanan et al., 2009) and dynamic cloudlets (Gai et al., 2016) reduce the aforementioned synchronization costs of VM-based approaches by detecting and leveraging nearby resources. However, such solutions are also beneficial and applicable to other MCC approaches and do not directly address the disadvantage of high synchronization costs. Yang et al. (Yang et al., 2014) mitigate the problem by analyzing the stack and heap to determine possibly accessed heap objects. Compared to CloneCloud, they achieve better execution times through reduced state transfer times. Nonetheless, still for each user a synchronized and customized VM is needed that runs at least the same operating system and application.

In contrast to the fine-grained VM-based approaches, the idea of Giurgiu et al. (Giurgiu et al., 2009) relies on modularized software. The developer has therefore to create his application by small software bundles that offer their interfaces as services and interact via services as well. Both the smartphone and the server have to run AlfredO (Rellermeyer et al., 2008) and OSGi (Alliance, 2009) which support modularized software written in Java. An offline profiling determines where to run the bundles by abstracting the resource consumption and the data flow of the interdependent bundles as a graph and cutting it in such a way that a given objective function is minimized or maximized. According to the authors, optimal cuts can be determined due to the small number of bundles. A downside of this approach is that an unfavourable modularization cannot even be compensated by an optimal cut when there exist just a few bundles. Despite that, the application's adaptation to OSGi may cause considerable additional efforts for software developers. Compared to VM-based approaches, components and their services may be reused by different customers.

Other works make use of well-known technologies like RPC (Balan et al., 2007) and remote services (Kemp et al., 2012) that are accessible via stubs. The main drawback of these approaches is the considerable effort a developer has to undertake to receive their benefits. In (Balan et al., 2007) the knowledge of the authors' description language Vivendi is required to create the tactics the developer wants to apply for the methods available via RPC. In (Kemp et al., 2012) developers have to create interfaces by using Android's interface definition language AIDL and to provide methods that transform their objects to so-called Android Parcels. Furthermore, a second implementation has to be supplied for the remote service.

Our MOCCAA framework makes use of Remote Method Invocation, but uses an intermediate serialization format that is translated seamlessly from/to Android as well as from/to Java representations. Delta synchronization is supported to reduce state transfer costs. Offloading is enabled through annotations at object and method granularity. VMs are not needed per user. This relieves application users as they do not need to search and lease own VMs. Instead, servers are the responsibility of the application providers. This allows them to reuse installed applications or even tailor them to a more efficient utilization of the available cloud resources. MOCCAA-RMS facilitates monitoring, scaling, managing and finding suitable resources within the application provider's domain. Monitoring and prediction costs on the mobile device are reduced through our analysis tool InspectA.

9 CONCLUSION AND FUTURE WORK

We proposed a configurable and extendable framework and architecture for Mobile Cloud Compu-

ting. In its simplest form, developers may use it by applying a few annotations and install the framework on a server. However, it can be extended with resource consumption prediction, a distributed and self-configuring Resource Management System (MOCCAA-RMS), and a grid-like computation pattern. In addition, we presented DRMI, a delta-synchronized RMI that is well suited for applications communicating deltas with their offloaded part. In future, DRMI will be equipped additionally with a sliding window approach that will be applied for arrays and Java Collections. Although the current DRMI version works for Java Collections and arrays, it can be further optimized for them if they only contain primitive data types. Additionally, we examine further patterns for splitting tasks and exploiting Cloud resources.

REFERENCES

Alliance, O. (2009). *OSGi Service Platform Service Compendium: Release 4, Version 4.2 Author: OSGi Alliance, Publisher: AQute Publishing Pages*. AQute Publishing.

Balan, R. K., Gergle, D., Satyanarayanan, M., and Herbsleb, J. (2007). Simplifying cyber foraging for mobile devices. In *Proceedings of the 5th international conference on Mobile systems, applications and services*, pages 272–285. ACM.

Bernstein, A. J. (1966). Analysis of programs for parallel processing. *Electronic Computers, IEEE Transactions on*, 5(5):757–763.

Chaumette, S., Grange, P., et al. (2002). Parallelizing multithreaded java programs: a criterion and its pi-calculus foundation. *Workshop on Formal Methods for Parallel Programming IPDPS*.

Chiba, S. (1998). Javassist - a reflection-based programming wizard for java. In *Proceedings of OOPSLA98 Workshop on Reflective Programming in C++ and Java*, volume 174.

Chun, B.-G., Ihm, S., Maniatis, P., Naik, M., and Patti, A. (2011). Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM.

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

Friedman, J. H. (1991). Multivariate adaptive regression splines. *The annals of statistics*, pages 1–67.

Gai, K., Qiu, M., Zhao, H., Tao, L., and Zong, Z. (2016). Dynamic energy-aware cloudlet-based mobile cloud computing model for green computing. *Journal of Network and Computer Applications*, 59:46–54.

Giurgiu, I., Riva, O., Juric, D., Krivulev, I., and Alonso, G. (2009). Calling the cloud: enabling mobile phones as interfaces to cloud applications. In *Middleware 2009*, pages 83–102. Springer.

Graf, J., Hecker, M., and Mohr, M. (2013). Using joana for information flow control in java programs-a practical guide. In *Software Engineering (Workshops)*, volume 215, pages 123–138.

Hall, M. A. (1999). Correlation-based feature selection for machine learning. *University of Waikato, New Zealand*.

Kaufman, L. and Rousseeuw, P. J. (2009). *Finding groups in data: an introduction to cluster analysis*, volume 344. John Wiley & Sons.

Kemp, R., Palmer, N., Kielmann, T., and Bal, H. (2012). Cuckoo: a computation offloading framework for smartphones. In *Mobile Computing, Applications, and Services*, pages 59–79. Springer.

Kohavi, R. and John, G. H. (1997). Wrappers for feature subset selection. *Artificial intelligence*, 97(1-2):273–324.

Kosta, S., Aucinas, A., Hui, P., Mortier, R., and Zhang, X. (2012). Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Infocom, 2012 Proceedings IEEE*, pages 945–953. IEEE.

Ou, S., Yang, K., and Liotta, A. (2006). An adaptive multi-constraint partitioning algorithm for offloading in pervasive systems. In *Pervasive Computing and Communications, 2006. PerCom 2006. Fourth Annual IEEE International Conference on*, pages 10–pp. IEEE.

Rellermeyer, J. S., Riva, O., and Alonso, G. (2008). Alfredo: an architecture for flexible interaction with electronic devices. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, pages 22–41. Springer-Verlag New York, Inc.

Satyanarayanan, M., Bahl, P., Caceres, R., and Davies, N. (2009). The case for vm-based cloudlets in mobile computing. *Pervasive Computing, IEEE*, 8(4):14–23.

Tridgell, A. (1999). *Efficient algorithms for sorting and synchronization*. PhD thesis, Australian National University Canberra.

Yang, S., Kwon, D., Yi, H., Cho, Y., Kwon, Y., and Paek, Y. (2014). Techniques to minimize state transfer costs for dynamic execution offloading in mobile cloud computing. *IEEE Transactions on Mobile Computing*, 13(11):2648–2660.

Zhao, Y., Hu, F., and Chen, H. (2016). An adaptive tuning strategy on spark based on in-memory computation characteristics. In *Advanced Communication Technology (ICACT), 2016 18th International Conference on*, pages 484–488. IEEE.