AIOLOS: middleware for improving mobile application performance through cyber foraging

Tim Verbelen[a], Pieter Simoens[a,b], Filip De Turck[a], Bart Dhoedt[a]

[a] *Ghent University – IBBT, Department of Information Technology, Gaston Crommenlaan 8 bus 201, 9050 Gent, Belgium*
[b] *Ghent University – IBBT, Ghent University College, Department INWE, Valentin Vaerwyckweg 1, 9000 Gent, Belgium*

Abstract

As the popularity of smartphones and tablets increases, the mobile platform is becoming a very important target for application developers. Despite recent advances in mobile hardware, most mobile devices fail to execute complex multimedia applications (such as image processing) with an acceptable level of user experience. Cyber foraging is a well-known computing technique to enhance the capabilities of mobile devices, where the mobile device offloads parts of the application to a nearby discovered server in the network.

Although first introduced in 2001, cyber foraging is still not widely adopted in current smartphone platforms or applications. In this respect, two major challenges are to be tackled. First, a suitable adaptive decision engine is needed to determine the optimal offloading decision, that takes into account the potentially high and variable latency between the device and the server. Second, an integrated cyber foraging platform with sufficient support for application developers is not publicly available on popular mobile platforms such as Android.

In this paper, we present AIOLOS, a mobile middleware framework for cyber foraging on the Android platform. AIOLOS uses an estimation model that takes into account server resources and network state to decide at runtime whether or not a method call should be offloaded. We also introduce developer tools to integrate the AIOLOS framework in the Android platform, enabling easy development of cyber foraging enabled applications. A prototype implementation is presented and evaluated in detail by means of both a chess application and a newly developed photo editor application.

Keywords: Distributed Systems, Cyber Foraging, Mobile Computing

1. Introduction

In the past decade, one has witnessed an increasing popularity of smartphones. According to Gartner this trend will continue, estimating an increase of 57.7% on smartphone sales in 2011 [1]. Due to advances in mobile hardware (e.g. display resolution and CPU power) and improved mobile network connectivity, new and more advanced services can be offered on mobile devices. The introduction of application markets on various mobile platforms has eased the publishing of applications, resulting in a myriad of mobile applications such as social and messaging clients, location-based services, games and many more. As applications and services become increasingly important, one can also witness a shift from mobile devices to ecosystems, such as Google's Android [2], with people no longer only buying a device, but an integrated combination of a device, a platform, applications and services.

Although the capabilities of mobile devices are increasing, they still cannot match their desktop counterparts, especially for complex multimedia applications such as image and video processing, object or face recognition and augmented reality applications. This encourages the use of cyber foraging to augment the capabilities of the mobile device by outsourcing CPU intensive parts of the application to remote servers [3]. These remote servers can be personal machines residing in the LAN network, as well as virtual machines in the cloud, where computing power is offered on demand [4], as shown in Figure 1.

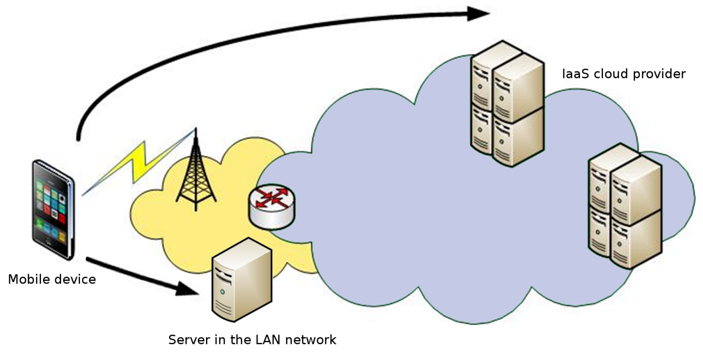


Figure 1: A mobile device's capabilities are enhanced by offloading to a server available either nearby or far away in the cloud.

Although considerate research effort has been spent on designing and developing cyber foraging systems [5][6][7][8], this paradigm is still not widely adopted in current smartphone application development. Two issues still need to be addressed. First, an adaptive offloading strategy is needed that copes with the limited bandwidth in wireless networks as well as with high and variable network latencies in a WAN environment [9]. Second, none of these cyber foraging systems are integrated in a mobile ecosystem, with sufficient developer support to ease the development of cyber foraging enabled applications. Only Cuckoo [10] does an effort offloading Android services and provides tools to assist in the build process, but the developer still has to define interfaces for computationally intensive parts and write both a local and a remote service implementation.

To address those two issues, we present AIOLOS[1]: a cyber foraging framework for Android. AIOLOS is built on OSGi [11], a module system and service platform in Java, which enables transparent monitoring and mobile code. For each method call the framework estimates the execution time for both local and remote execution based on the argument size, and then chooses the appropriate execution location. Afterwards, the estimation parameters are updated, taking into account the new profiling data and changing network parameters. In this way the system will optimize the application execution, taking into account the device capabilities and the network connectivity, without putting an additional burden on the application developer.

The second issue is addressed by integrating AIOLOS in the Android platform and providing the developer support for creating cyber foraging enabled applications. We present a plugin for the Eclipse IDE, the most popular Android development environment, that allows the developer to annotate possible offloadable classes. At build time the plugin generates OSGi bundles that publish the annotated classes as OSGi services, and an OSGi runtime with the AIOLOS framework is embedded in the application.

To evaluate the AIOLOS framework we used the Eclipse plugin to enable cyber foraging for an existing chess application on Android. We also developed a photo editing application to show the effectiveness of the estimation model.

The remainder of this paper is structured as follows. Section 2 gives an overview of related work on code migration and cyber foraging. Section 3

[1]*aiolos* is an ancient greek word meaning "quickly changing, adapting"

describes the algorithm for deciding when to offload and Section 4 shows how we estimate performance from a history based profile. Section 5 describes the architecture of our offloading middleware. In Section 6, we discuss implementation details of how AIOLOS is integrated in Android, and in Section 7 we present the programming model and developer tools to ease application development. In Section 8 we evaluate the framework using relevant use case scenarios. We conclude this paper in Section 9 and discuss the limitations and open problems of our approach.

2. Related work

The idea of using remote servers, also called surrogates, to augment the processing capabilities of mobile devices was first proposed by Satyanarayanan [3] as cyber foraging. Early cyber foraging systems such as Spectra [5] and Chroma [12] use a tactics based scheduler to pick methods to outsource taking into account history-based resource predictions [13]. The most important drawback of these first systems is that they rely on pre-installed remote procedure calls, implying that all remoting has to be programmed by the application developer. To facilitate application development the Vivendi tactics description language and stub generator were proposed [14], where the application developer declares a tactics file declaring remote operations, their parameters and fidelities from which stubs are generated. With AIOLOS, the developer marks offloadable methods using source code annotations, which avoids the need for a complex language and syntax.

Later systems make use of mobile code, where the system partitions the software at runtime and migrates code to the server, with minimal interference from the developer. Several systems exist, offloading either at a class, method or software component level.

Gu et al. [15] present an adaptive offloading framework for offloading Java classes, using a fuzzy control model. Ou et al. [16] also use class outsourcing, in combination with a (k+1) partitioning algorithm. Because applications typically consist of hundreds to thousands of classes, extensive monitoring and computation is needed to identify classes to offload, making this approach less suited to adopt at runtime.

MAUI [7] uses extensive profiling to decide to offload method calls on the Microsoft .Net runtime environment. The goal is to optimize the energy usage, and an Integer Linear Programming (ILP) approach is used to optimize the deployment. The goal of our approach is to optimize the execution time, and ILP is not used as it is too time consuming to solve for each method call.

The Scavenger cyber foraging system [6] outsources Python methods with a scheduler using history-based profiling. Similar to AIOLOS, Scavenger requires the application developer to annotate methods that are candidates for remote execution. Based on these annotations, the system takes into account the size or the value of the input parameters of the method call to estimate the execution time at runtime (e.g. an image processing filter will take longer on a larger image). Scavenger however does not support callbacks and assumes a constant network latency which is statically defined per device, whereas in AIOLOS, the network latency is estimated from previous remote method calls.

Zhang et al. [17] propose a mobile code framework where platform independent software components – called weblets – can be outsourced to the cloud based on a Bayesian learning scheduler. Giurgiu et al. [18] and Verbelen et al. [19] use OSGi components to build a graph model of the software and use graph cutting algorithms to distribute these components. Han et al. [20] present a flow-based algorithm to partition software components, which is evaluated by simulation. The drawback of these component based approaches is that a software component will not continuously generate the same load, potentially leading to instable partitionings.

Kemp et al. [10] present Cuckoo, an offloading framework for Android that is able to offload Android services. An extra build step is introduced to make an existing Android application capable of offloading its service components. However, it is still up to the developer to choose what to execute remotely. The approach also does not support callbacks from the offloaded code.

As surrogates for remote execution, Goyal et al. [21] propose the usage of virtual machine technology. Because the deployment of virtual machines in a cloud can lead to high WAN latencies, Su et al. present Slingshot [22], where the VMs are co-located with the wireless access point. Satyanarayanan et al. [9] also place the surrogates, called “cloudlets”, in the physical proximity of the mobile user. Chun et al. [23] use virtualization to create a clone of the mobile device platform to be able to migrate parts of mobile applications without modification.

In this paper we describe AIOLOS, a cyber foraging framework that combines the approaches at both component and method level. On the one hand software components are used as unit of deployment to offload, which keeps the overhead to monitor at runtime low as applications only consist of tens of components. On the other hand, offloading decisions are made for each method offered by the components and the framework will select methods to offload. The execution time of method calls is estimated based on the size of their arguments, also taking callbacks into account. The network characteristics are estimated from the duration of previously monitored remote method calls, using a random sample consensus algorithm.

To facilitate the development of cyber foraging enabled applications, we provide developer tools that integrate the AIOLOS framework in Android applications. The developer only has to annotate offloadable classes, and an extra build step will refactor the source code and incorporate AIOLOS in the application, which makes it easy to use AIOLOS in new as well as in existing Android applications.

3. Decision algorithm for offloading method calls

When a method of an annotated class is called, a decision has to be made whether to execute the method call locally or on a discovered remote surrogate. The two most important objectives to optimize are execution time [6] and energy [7]. We present two decision models that can be used in AIOLOS to optimize each of these objectives.

3.1. Optimize execution time

When optimizing the execution time, the optimal decision is to call the remote instance when the remote execution time is smaller than the local execution time, or as depicted in Figure 2 if $T_{remote} < T_{local}$.

In the following discussion we assume that callback methods do not need much CPU resources nor send much data compared to the other service methods. This assumption holds if callback functions are just handling small intermediate updates of the GUI. Figure 2 shows an example offloading scenario, where A performs a remote call of B. If C is called by B, it is preferable to also execute this component on the server, because the cost of the callback from C to A is assumed small. If the callback method takes large input arguments, it could be preferable to also keep C local. However, this solution can only be found using a graph cutting algorithm, which causes significant overhead to calculate before every service method call [8].

To estimate the local execution time T_{local}, a history based profile is maintained for each service method, as described in Section 4. To estimate the

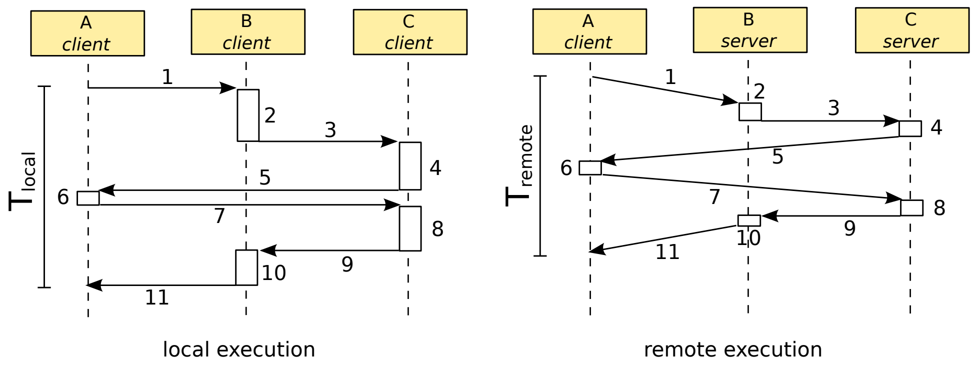


Figure 2: Time needed for local execution T_{local} (on the left) and for remote execution T_{remote} (on the right).

remote execution time T_{remote} more information is needed. First, we introduce the speedup factor α of the processing time of the remote method calls, as the server's CPU is likely to be faster than the processor of the mobile device (2, 4, 8 and 10 in Figure 2). Two parameters β and γ represent the network bandwidth and latency, and are used to estimate the network round trip time as a linear function of the bytes sent for the remote call and callbacks (1,11 and 5,7 in Figure 2). The processing time of the callback (6 in Figure 2) remains the same, as a callback is always executed locally. The formula to estimate $\hat{T}_{remote}$ thus becomes:

$$\begin{aligned}\hat{T}_{remote} &= \frac{1}{\alpha} \times \sum_{i \in RM} (\hat{T}_{CPU,local_i}) + \frac{1}{\beta} \times (A + R) + \gamma \\ &\quad + \sum_{j \in CM} (\hat{T}_{CPU,local_j} + \frac{1}{\beta} \times (A_j + R_j) + \gamma)\end{aligned}$$

With RM the collection of remotely executed methods, the estimated local processing times $\hat{T}_{CPU,local_i}$ are summed and divided by the speedup factor α. The network time of the remote method call is estimated using β, γ and the argument size A and estimated return size R. Finally, CM defines the collection of callbacks, for which the local processing time is added, as well as an additional estimated round trip time. Parameters α, β and γ are determined using history information of local and remote executions, as described in Section 4.

3.2. Optimize energy

When optimizing the energy usage, the method should be offloaded if energy would be saved by remote execution. As stated by [24], a simple decision model is to offload if the energy consumed by sending and receiving bytes to and from the server is smaller than the energy saved by offloading the computation, thus to offload when

$$\begin{aligned}\hat{E}_{saved} &= E_{CPU} \times \sum_{i \in RM} (\hat{T}_{CPU,local_i}) - E_{TR} \times A - E_{RCV} \times R \\ &\quad - \sum_{j \in CM} (E_{RCV} \times A_j + E_{TR} \times R_j) \\ &> 0\end{aligned}$$

In this case, the decision will depend on the energy model of the mobile device, where E_{CPU} denotes the energy consumed per time unit by the CPU, and E_{TR} and E_{RCV} respectively denote the energy cost for transmitting and receiving a byte. However, it is difficult to estimate these parameters, which are hard to measure using only software tools, and depend on the hardware and the hardware state [17]. Therefore, in the remainder of this paper we focus on optimizing execution time.

4. Parameter estimation using history based profiles

4.1. Estimating T_{local} and R

To estimate the remote execution time with the formula presented in Section 3, first the local execution time T_{local} and the return size R of the method need to be estimated. This is done using a history based profile, which can be built at runtime, or can be provided from previous sessions. In [13] it is shown that depending on the application, the load can be accurately predicted from history logs, especially when application specific knowledge about the input parameters is available. In this work, we assume to have no a priori knowledge on the application, and the only parameter we can use to predict the execution time of a method is the size of the input parameters.

For each method call that is executed locally, the framework records the size of the arguments, the size of the returned result and the local execution time. The next time the method is called, the local execution time and the size of the result are estimated by looking up the closest data points (with respect to argument size) in the history based profile, and by linearly interpolating between these values. To limit the size of the profile and to speed up data lookup in the profile, close data points are clustered in buckets, for which only an average value is kept, as described in [6].

When the argument size of the method does not relate to the execution time, we will end up with a profile containing average execution times for each argument size, which will converge to the same value for each argument size: the average execution time of the method.

4.2. Estimating α, β and γ

Parameters α, β and γ can be determined using history information of local and remote executions. The speedup factor α is the ratio of the processing time of local executions and the processing time of the method call on the server. Parameters β and γ can be estimated as the measured bandwidth and latency of the network. In previous work [13] [7] the network characteristics were estimated using a moving average of the network transfer time. The main disadvantage is that a single outlier measurement, although averaged, can quickly reduce the quality of the predicted network transfer time.

In order to reduce the effect of outlier measurements, we estimate the parameters α, β and γ from k previous remote method calls using a random sample consensus (RANSAC) algorithm [25], which is commonly used in computer vision, but is also useful in other estimation problems. The algorithm will pick a subset of the input data as hypothetical inliers, and fit a model to those. All other data is then fitted to this model, and if datapoints fit the model well, they are added to the inlier subset. Finally, the model is re-estimated using all of the found inlier data points, and the error of the model to the inliers is calculated. This proces is repeated a number of iterations and the model with the smallest error is returned.

The input data to estimate α is the ratio of the (estimated) local processing time and the (measured) remote processing time of the last k remote method calls. In each iteration, the algorithm randomly selects one ratio value and the absolute difference with the other values is calculated. The other values for which this difference is beneath a predefined threshold are marked as inliers, and the resulting α is the average of this inlier set.

To estimate β and γ, for each of the last k offloaded method calls the time needed for sending $(A + R)$ bytes is calculated as the difference of the remote time measured at the mobile device and the execution time measured at the server. Iteratively two points are randomly picked and a line is fitted through these points. All other points are classified as inliers if the error to the line is beneath a certain threshold (i.e. 10%). Finally, a linear regression is calculated over the set of inliers. This enables robust estimation of the model parameters without errors caused by outliers.

5. AIOLOS architecture

First, relevant background information on the Android platform and its components is provided, and the use of OSGi is motivated. Next, the architecture of the AIOLOS cyber foraging framework is presented.

5.1. Android

Android is an open source platform developed by the Open Handset Alliance targeted at smartphones, tablets and other devices with limited resources. Android applications are written in Java and compiled to Dalvik bytecodes, which run on the Dalvik Virtual Machine. Android applications are composed of different components: Activities, Services, Content Providers and Broadcast Receivers.

An Activity provides the basic interaction logic with the user, containing a user interface and offering some basic computing capabilities. An Android Service is a component that runs in the background, mainly used for long-running background processes, e.g. playing music or fetching data without blocking the user interface. Other components can bind to a Service and communicate with the Service through inter process communication (IPC). Content Providers are used for managing a shared set of application data. Finally, Broadcast Receivers are small components that respond to system-wide broadcast announcements e.g. an announcement that the battery is low.

5.2. OSGi

OSGi [11] is a service oriented module management system in Java allowing to dynamically load and unload software modules – called bundles – at runtime. OSGi bundles can expose a service interface by registering an implementation of this interface with the OSGi service registry. When a bundle wants to call another service, it queries the service registry for available implementations. The portability of Java enables the execution of the same code on different platforms and architectures, facilitating remote execution

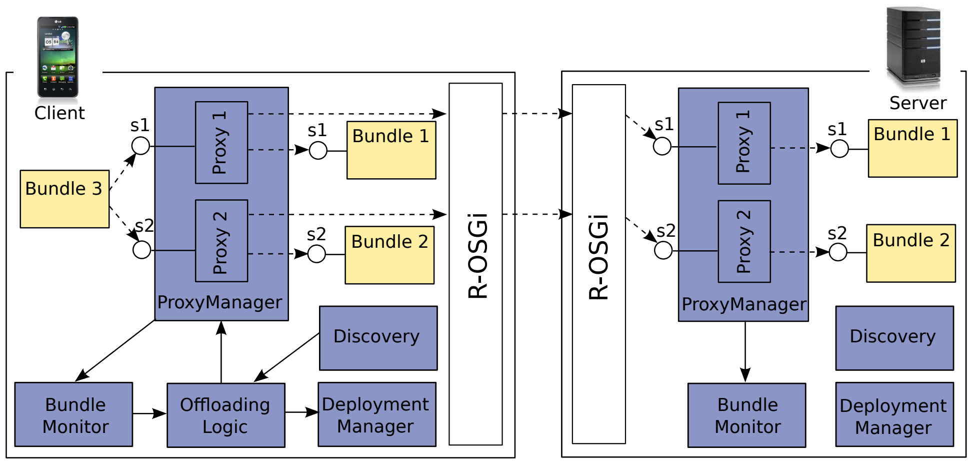


Figure 3: Overview of the offloading middleware. The Proxy Manager creates proxies for the registered services (s1 and s2) of the application bundles. These proxies can switch between local and remote method execution and monitor all method calls. The Bundle Monitor aggregates this monitor information to build profiles of all methods called. This information is used by the Offloading Logic to instruct the Deployment Manager to outsource bundles, or to update the proxy policies when to offload.

and code migration. Because OSGi was first designed for embedded devices, it is also lightweight.

The biggest hindrance for the straightforward use of OSGi in this context, is the fact that the developer has to build his application as a collection of OSGi bundles, which is not trivial, especially not on the Android platform. To cope with this issue, we present extra tools for the Eclipse IDE in Section 7 that convert a standard Android application into an OSGi enabled application, with a minimum of effort for the application developer.

5.3. AIOLOS architecture

The architecture of the AIOLOS framework is shown in Figure 3. Both the client and the server run an OSGi container with AIOLOS bundles and application bundles. R-OSGi [26] is used for remote method invocations across OSGi platforms.

Each application bundle's service interface contains a number of service methods, which will be monitored and profiled at runtime by the framework. Using this method profile AIOLOS decides for each method call whether it will be executed locally or remotely.

The middleware is composed of the following OSGi bundles:

1. The **Proxy Manager** is responsible for creating and managing proxies for each application bundle's services. By proxying each registered service, the middleware can on the one hand decide to forward a service call to a remote service implementation rather than to the local one, and on the other hand gather monitoring information of all service calls. In order to proxy services, the Proxy Manager uses Service Hooks, introduced in version 4.2 of the OSGi specification [11]. When an application bundle registers a service with the OSGi framework, the Proxy Manager gets a callback through the Listener Hook, and generates a proxy for this service which will also register the service with the OSGi service registry. When a bundle looks up the service in the OSGi service registry, the Proxy Manager can hook in using the Find Hook and return the service registered by the proxy, hiding the original service implementation. In this way, all monitoring and remote execution happens transparently to all application bundles. This also allows to handle errors transparently, for example when the network connection is lost during remote execution, a time-out is detected by the proxy and the method can be executed locally.
2. The **Bundle Monitor** gathers monitor information of all application bundles. When a service method is called, the Bundle Monitor gets notified by the involved proxy and captures the size of the input arguments, the size of the return value and the execution time of the method call. Using this information, an execution profile for each method is constructed, which is used to identify candidate methods for remote execution by the Offloading Logic bundle.
3. The **Offloading Logic** implements the algorithm to decide whether to offload a method or not. This component periodically checks whether the estimated execution times match the actual monitored values provided by the Bundle Monitor, and if necessary parameters α, β and γ are recalculated as described in Section 4. The Offloading Logic will also instruct the Deployment Manager to copy bundles to the server when methods are identified that would benefit from offloading, in case the associated bundles are not available at the server yet. Because of the component based architecture of AIOLOS, this bundle can be easily replaced by a bundle implementing another decision strategy, e.g. for optimizing energy.

4. The **Deployment Manager** can migrate bundles to remote servers by sending the bytecodes to the server and installing the bundle and its dependencies there on the OSGi runtime.

6. AIOLOS and Android integration

To integrate AIOLOS and Android, an OSGi container has to be deployed on the Android platform. One solution, proposed and implemented by ProSyst [27], is to create an Android OSGi container application, which can be executed as a regular Android app. Developers code their application as a number of OSGi bundles, which can be deployed and started in the OSGi container. However, there are a number of drawbacks to this approach. First, a clear separation remains between Android applications and OSGi applications, as the OSGi container has to be started first before one can access the OSGi applications. Second, this forces the developer to create OSGi bundles instead of a regular Android application, and to rely on an SDK provided by ProSyst to access Android internals.

Therefore, we choose to embed an OSGi container within the Android application, and only package offloadable components as OSGi bundles as depicted in Figure 4. In order to let the Android application communicate with the bundles, the bundle's services can be looked up in the OSGi container. The Android application can also register callback services with the

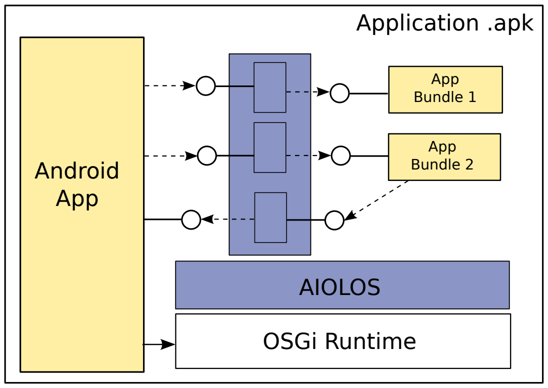


Figure 4: An OSGi container is embedded within the Android application. Offloadable parts of the Android application are packaged as OSGi bundles and deployed in the embedded OSGi container where the AIOLOS framework handles the cyber foraging functionalities. The Android application can lookup services through the OSGi runtime, and can also register callback services with the OSGi runtime.

OSGi runtime, so the OSGi bundles are able to make callbacks. In order that the offloadable components can be executed both on the Android device and on a remote server, the code is compiled in both the Dalvik bytecode format and the regular Java bytecode format.

The resulting application acts as a regular Android application, and no additional SDK is needed. However, a considerate burden is put on the application developer when he wants to develop an AIOLOS enabled application. The following steps are required to convert a regular Android application to an offloadable application:

1. Identify parts of the application suitable to offload.
2. Create a separate OSGi project for each offloadable part.
3. Identify and create service interfaces that will act as access points to the bundle.
4. Identify possible callback interfaces that allow the bundle to perform certain callbacks to the rest of the application.
5. Write the necessary OSGi code that manages the lifecycle of the bundle. This includes correct initialization of the bundle, registration of the bundle's services, searching for dependent services, etc.
6. Build and package each OSGi bundle, including a correct bundle manifest describing the OSGi bundle and its dependencies.
7. Include all the resulting OSGi bundles as resources in the Android project.
8. Embed an OSGi runtime in the Android application, that manages all the generated OSGi bundles, together with the AIOLOS framework bundles.

This task can be tedious and error prone, and requires a lot of knowledge about the OSGi specification and APIs. To simplify this process, we developed a plugin for the Eclipse IDE that automates these steps as much as possible and reduces the task of the application developer to step 1: identifying offloadable parts of the application using annotations which is discussed in Section 7.

7. Development tools for AIOLOS

In order to identify offloadable parts of the application, the developer can use Java Annotations to indicate classes that should act as services for offloadable components. There are a few rules the class must adhere to in order to enable cyber foraging.

1. The class needs a no-argument constructor to be present, either explicitly or implicitly, to enable the framework to create class instances.
2. All arguments and return values used in the public class methods should be serializable, to be able to send these values over the network.
3. The class should not have dependencies to the Android APIs, as these are platform specific and are likely not present on the surrogate.
4. At this moment only stateless services are supported for offloading, so all state needed should be passed as a method argument.

When one of these rules is violated, the build tool will show an error or warning message indicating the problem, if possible hinting the developer how to fix the issue (e.g. by indicating which argument type should be made serializable).

An example offloadable class is given in Listing 1. The `Worker` class has one stateless public method `doWork`. In order to work the `Return` class should be serializable.

```
@Offloadable
public class Worker {
  public Return doWork() {
    // do something
  }
}
```

Listing 1: Code snippet of a class annotated @Offloadable.

When the builder finds an annotated class that adheres to the rules, a service interface is extracted from this class. This service interface is then registered with the OSGi runtime at startup, and an implementation object is bound to this interface. The extracted service interface and its implementation are given in Listing 2.

```
public interface WorkerService {
  public Return doWork();
}

public class Worker implements WorkerService {
  public Return doWork() {
    // do something
  }
}
```

Listing 2: After refactoring a service interface is extracted from the offloadable class.

Next, all method calls to the offloadable class are automatically refactored. Instead of invoking the method on a known reference to this object, first a reference will be looked up in the OSGi registry. Suppose for example that the application contains a button that executes the `onWork` method when the button is clicked, a sample `onClick` method is given in Listing 3.

```
// method called when clicked on button
public void onClick(){
  Worker w = new Worker();
  Return r = w.doWork();
  // print result on screen
}
```

Listing 3: Code snippet of a button onClick method calling the offloadable class.

Listing 4 shows the refactored `onClick` method. A reference to an implementation of the `WorkerService` interface is looked up in the OSGi registry. The OSGi runtime is wrapped in a singleton class so that the object is accessible from anywhere in the application.

```
public void onClick(){
  WorkerService w = null;
  w = OSGiRuntime.getInstance().lookupService(WorkerService.
      class.getName());
  Return r = w.doWork();
  // print result on screen
}
```

Listing 4: Refactored method call to the offloadable class. The reference is looked up with the OSGi Runtime.

Because the offloadable class has to be executable on a surrogate, it cannot contain dependencies to classes of the Android APIs, as these are specific to the Android OS, and hence indeed not executable at the surrogate. This type of calls are often system calls to the mobile device hardware. However, dependencies of the offloadable class can still invoke system calls. If this is the case, a callback service interface is extracted and registered with the OSGi runtime. As the system calls can only be executed on the device itself, this callback service cannot be outsourced.

Next, for each offloadable class an OSGi bundle is formed based on the class dependency graph. Classes depending on a single offloadable class are added to the OSGi bundle of the offloadable class. Classes that are shared between multiple bundles are put in a separate OSGi bundle. An example is shown in Figure 5, where a dependency graph of an application's classes is given. Classes A, G and J are dependent on the Android SDK, and therefore are put in the Android application part that cannot be outsourced. Classes C and H are marked as offloadable, thus two application bundles will be formed. Because classes D and E are only dependent on C, they will be placed in the application bundle with C. The same goes for I that is put in the application bundle with H. Classes B and F are shared between multiple bundles and the Android dependent code, and therefore these bundles are put in a shared bundle. Lastly the tool discovers a dependency of H to J, which on its turn depends on the Android SDK. This means that J is marked as a callback class from which a callback service is formed.

The build process starts with generating a bundle activator class for each OSGi bundle, that manages the lifecycle of the bundle and registers the bundle's services with the OSGi runtime at startup. From the class dependency graph also a correct bundle manifest is created, describing the resulting OSGi bundle and its dependencies.

Finally, code is added to launch a Felix [28] OSGi runtime and AIOLOS framework together with the generated OSGi bundles, and to set up all service registrations when the Android application is started. The remain-

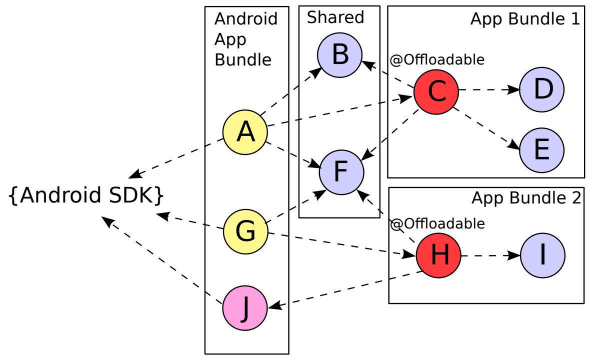


Figure 5: Bundles are formed from the class dependency graph. Classes C and H are marked as offloadable by the developer, and will form application bundles with the classes that only depend on them. Classes A, G and J depend on the Android SDK and will be put in the Android application bundle that cannot be offloaded. Moreover J will be marked as a callback class from which a callback service is formed. Classes B and F are shared between the different bundles and are packaged as a shared bundle.

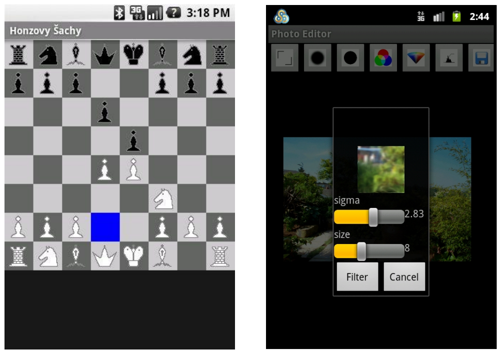
ing Android application code is built and bundled with the OSGi runtime, AIOLOS and the generated bundles in an Android package (.apk) file that can be installed on the Android device.

8. Evaluation results

We evaluate the effectiveness of mobile offloading using two example applications, shown in Figure 6. The first one is Honza's Chess, an existing open source chess game developed for Android [29]. The second application is a photo editor application that enables the user to perform several image filters such as blur, sharpening or histogram equalization on his photos. When the user clicks an image, he first sees a preview window enabling him to adjust the filter parameters. When the parameters are correctly set, the image filter is performed on the whole image.

Both applications are developed as regular Android applications and can also run as such. By using the annotations and Eclipse IDE tool presented in Section 7 these applications are integrated in the AIOLOS framework enabling to offload certain operations.

To show the effectiveness of mobile offloading, we have conducted experiments on two Android devices: a Motorola Milestone, powered by a 600 MHz ARM Cortex A8 processor running Android 2.2, and an LG Optimus 2x, powered by a dual-core 1 GHz ARM Cortex A9 processor connected to a WiFi access point. To offload parts of the application, we used a quad core server clocked at 2.4 GHz deployed close to the mobile access point in the



(a) Honza's Chess (b) Photo Editor

Figure 6: Two Android applications running with the AIOLOS framework. On the left Honza's Chess [29] and on the right a Photo Editor application.

LAN network, as well as a 2.6 GHz quad core server deployed in the Amazon EC2 Cloud.

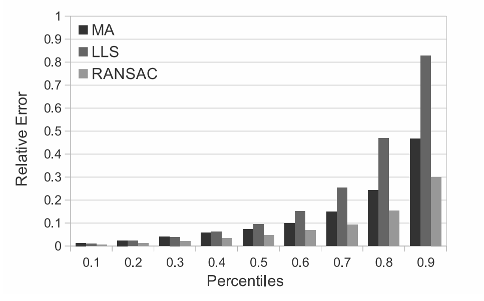
8.1. Estimation accuracy

To evaluate the estimation accuracy we compare three methods for estimating the parameters α, β and γ. The first method is a moving average (MA) method where for each remote method call a new value is measured for α, β and γ as $new = p \times (measured) + (1 - p) \times (old)$, with $p = 0.75$ for α and γ and $p = 0.875$ for β, as proposed in [30]. The advantage of this method is that the estimated value quickly adapts to context changes, but the disadvantage is that outlier measurements have a bad effect on the estimation accuracy. The other two methods use the measurements of the latest k remote method calls. The linear least squares (LLS) method calculates the average over the k values of measured speedup factors α and a linear least squares fit results in β and γ. The RANSAC method is the approach presented in Section 4.

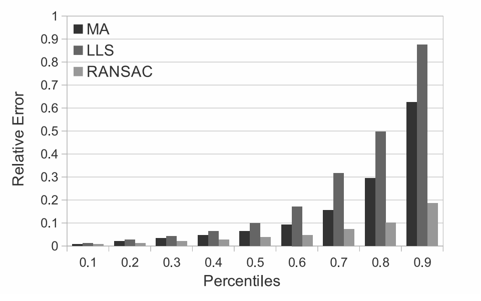
The three methods are compared in the following experiment using the photo editor application on the LG Optimus 2x. We execute randomly a sharpen filter, a color filter or a histogram equalization operation on a generated image with a width and height randomly selected between 100 and 1500 pixels. First we execute 100 operations locally in order to have some history to estimate the local execution time. Next, 200 method calls are executed remotely on the server in the LAN network. After each method call the measured remote execution time is compared to the estimated execution time and the parameters α, β and γ are updated. The 0.1 to 0.9 percentiles of the relative error for each of the three strategies, using a history of $k = 5, 10$ and 15 for the LLS and RANSAC methods are shown in Figure 7.

Although the LLS method uses more history for estimation, it performs worse than the MA method. This is because an outlier measurement only affects one estimation in the MA method, while it affects k estimations in the LLS method. The RANSAC method outperforms the MA method, as it tries to remove the effect of outliers in the measurements. When the amount of history k increases, the RANSAC method becomes better, but the gain becomes smaller as k increases: when increasing k from 10 to 15 the gain is negligible.

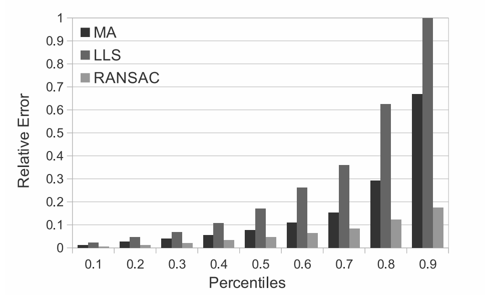
Our experiments show that the RANSAC algorithm with $k = 10$ produces estimations within 10% of the measured values in 80% of the cases, and adapting to a new context after five method calls. Figure 8 shows the



(a) k=5



(b) k=10



(c) k=15

Figure 7: The 0.1 to 0.9 percentiles of the relative error of the estimated remote execution time using a moving average (MA), a least linear squares (LLS) and a RANSAC estimation strategy. The LLS and RANSAC use a history of $k = 5, 10$ and 15 calls. RANSAC offers the best results for larger history k.

estimated execution time versus the measured execution time for the blur operation as a function of the argument size. The outliers present in the actual execution times do not influence the estimation parameters, as they are filtered out by the random sample consensus algorithm. As the argument size increases, there is more spread in the values because of the increase in network traffic.

Next, we also evaluate how the estimators react to a changing context. Therefore, we conduct the same experiment, but we limit the bandwidth using TC and Netem [31] on the Linux server. First we limit the bandwidth to 10 Mbit/s and at a certain moment in time, we reduce the available bandwidth to 5 Mbit/s, which could reflect a change in network conditions (e.g. a switch from WiFi to 3G). Figure 9 shows how parameter β changes over time for the three methods. Because the MA method only uses the latest measurement, it already adapts to 5 Mbit/s after one method call. The LLS

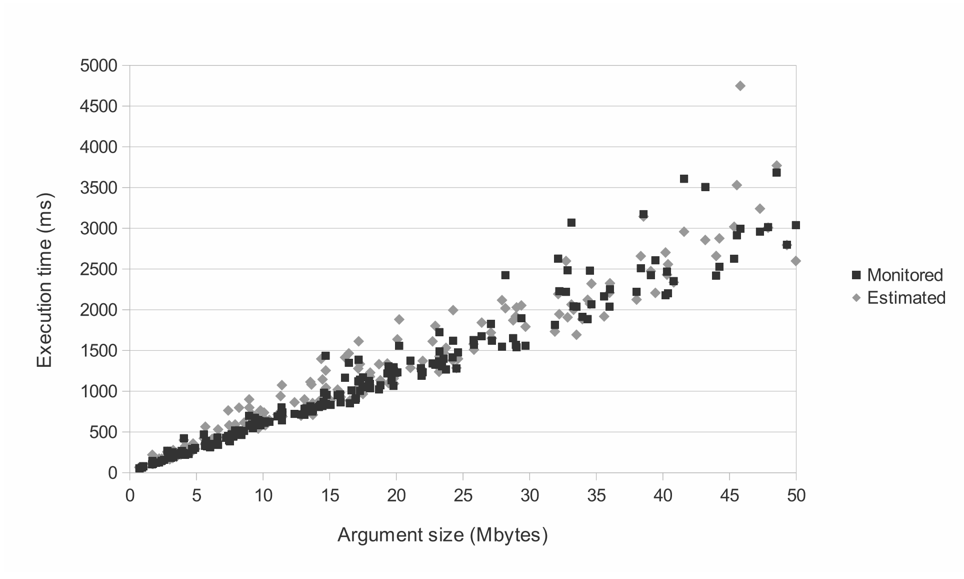


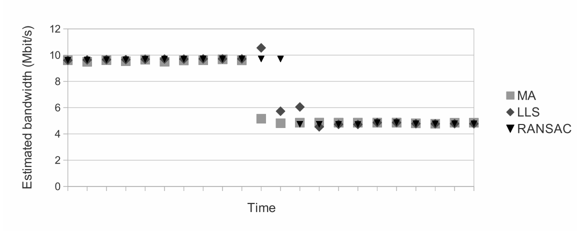
Figure 8: Estimated versus the actual monitored execution time of a blur operation on images of different size, using the RANSAC algorithm with $k = 10$.

approach uses all of k measurements, and thus will need k method calls before adapting to the new bandwidth. The RANSAC method will choose the value with most inliers, and thus will need $k/2$ method calls before adapting.

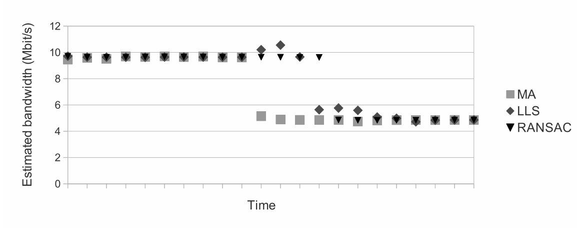
Table 1 shows the different values for parameters α, β and γ when offloading to a remote server in the LAN network or to the cloud from the Motorola Milestone and the LG Optimus 2x. The values for parameter α show that the server in the cloud is around 25% faster than the one in the LAN network, and the Optimus 2x is around 3 times faster than the Milestone. The values for parameters β and γ are about the same for both devices and strongly depend on the server location.

Table 1: Values of the parameters α, β and γ for both the Motorola Milestone and the LG Optimus 2x when offloading to a server in the LAN network and to the Amazon EC2 cloud.

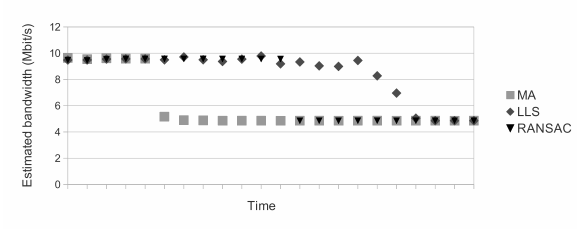
		α	β (Mbit/s)	γ (ms)
LG Optimus 2x	LAN	3.35035	14.50	21.7
	Cloud	4.04153	6.284	469.2
Motorola Milestone	LAN	11.1052	12.05	3.5
	Cloud	14.1115	6.509	416.5



(a) k=5



(b) k=10



(c) k=15

Figure 9: The ability to adapt to a bandwidth change in time using a moving average (MA), a least linear squares (LLS) and a RANSAC estimation strategy. The LLS and RANSAC use a history of $k = 5, 10$ and 15 calls. The MA approach adapts after one method call, the LLS and RANSAC need respectively k and $k/2$ method calls before the bandwidth change is taken into account.

8.2. Honza's Chess

For the chess application, we annotated the class that calculates the next move of the AI player. The AI algorithm conducts a tree search to find the best move. To limit the calculation time we thresholded the maximum depth of the search tree. We executed the first 8 moves of a chess game and compared the execution time of the AI algorithm executed on both devices locally, as well as offloaded to the nearby server and the cloud.

The experiment was repeated 10 times and the average execution times are shown in Figure 10. Offloading to the server in the LAN network always performs best. Offloading to the cloud also outperforms local execution on the Milestone, but the Optimus 2x can sometimes be slightly faster. The reason the cloud execution is 1.5 to 2 seconds slower than the server in the LAN network, is that the Chess implementation also performs screen updates giving the user feedback about intermediate best found solutions, resulting in callbacks during remote execution. These callbacks involve a communication over a relatively high delay WAN network, thereby contributing to the remote execution time. On average, offloading to the LAN network leads to a 80 to 90% increase in performance, and offloading to the Amazon EC2 Cloud offers

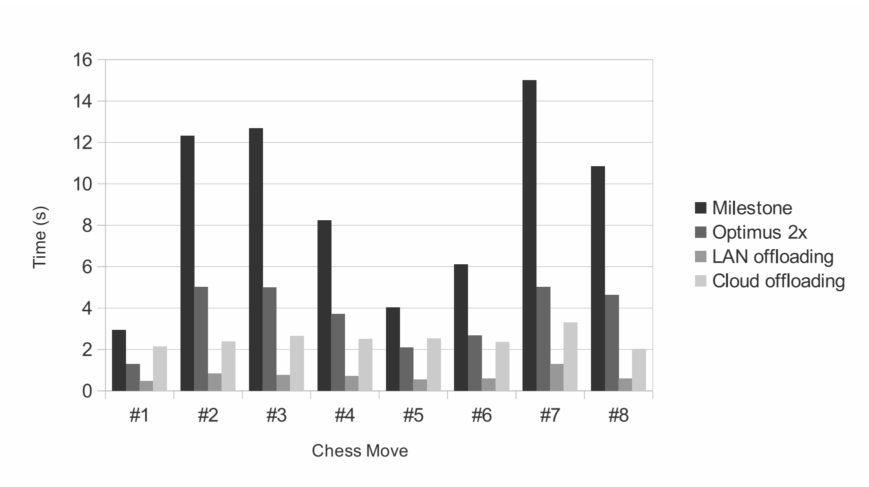


Figure 10: Execution time of the Chess AI algorithm for different chess moves for local execution on both the Motorola Milestone and the LG Optimus 2x, as well as remote execution on a remote server deployed in the LAN network and one deployed in the Amazon EC2 Cloud. Error bars are omitted as standard deviations are too small to be visible.

a 20 to 65% benefit, depending on the device and thus in this case AIOLOS will always choose to offload.

One can also see a high fluctuation in the local execution times, while the remote execution times remain more or less constant. This is due to the fact that for local execution the task is CPU bound, and the calculation time will depend on the number of possible moves to consider. As the machines at the server side are much faster, here the task becomes data bound, and the amount of data sent over the network will determine the execution time. Also note that if the player wants to increase the difficulty, he would increase the calculation depth threshold, resulting in higher calculation times and even more gain of offloading.

In this application, the execution time will depend on the pieces on the chess board, rather than the argument size. Therefore, AIOLOS will not be able to accurately predict the local execution time, but rather return an average execution time. However, for this use case this is sufficient to take the correct decision and to offload the method.

8.3. Photo Editor

For the Photo Editor application, all image filters are annotated for possible remote execution. We performed an experiment using the sharpening filter, which is one of the more complex implemented filters. Again we measure the local execution time on both mobile devices and the remote execution time when offloaded to a server in the LAN network and one in the cloud. The sharpening filter is executed 30 times on three different image sizes: 500×500, 1000×1000 and 1500×1500 pixels. The images are raw RGB images, thus a large amount of data has to be sent back and forth in the case of remote execution.

The average execution times are shown in Figure 11, together with the error bars indicating the standard deviation. For the Milestone, remote execution is always more beneficial than local execution. For the 1500×1500 image, we omitted the bar for the Milestone, as the execution time is over 30 seconds. The Optimus 2x is faster than execution on the cloud, but benefits from remote execution on the server in the LAN network as the image size gets higher (e.g. for the 1500×1500 remote execution is 23% faster). In this case, as AIOLOS uses the input size of the method call to estimate local and remote execution time, the method call will only be offloaded when the image size is large enough to offer a significant gain when executed remotely.

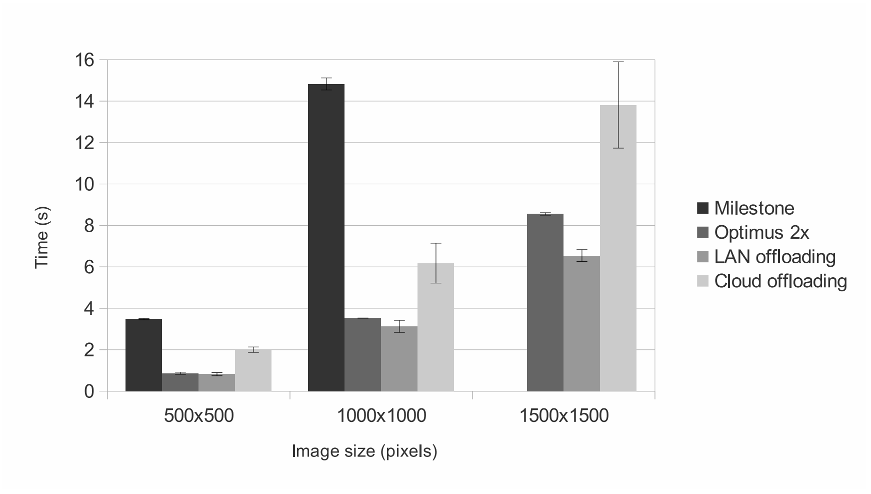


Figure 11: Execution time of a sharpening filter on a 500×500, 1000×1000 and 1500×1500 RGB image for local execution on both the Motorola Milestone and the LG Optimus 2x, as well as remote execution on a remote server deployed in the LAN network and one deployed in the Amazon EC2 Cloud. In the case of the 1500x1500 image local execution on the Milestone is omitted for the clarity of the figure and was more than 30s.

Also note that the standard deviation increases with the latency to the remote server. In order to make the application more suitable for (cloud) offloading, one could try to lower the data communication, e.g. by sending the JPEG images instead of the raw images.

8.4. AIOLOS overhead

The dominant overhead of AIOLOS is introduced when the application starts up for the first time. Depending on the application, three to five seconds are added to the startup time, measured on the LG Optimus 2x, to initialize the OSGi framework and to install and setup all the framework and application bundles. Each method call of an offloadable method passes through a proxy, which will record the execution time and calculate the size of the arguments and return value. This causes a few additional milliseconds of overhead per method call, which is negligible in the presented use cases. A history based profile of previous local method calls is maintained for each offloadable method, and the size can be limited by a predefined threshold. For the photo editor application 20 to 30 local method calls were already sufficient to accurately predict the local execution time.

8.5. Development tools

The development tools described in Section 7 greatly simplify the development process. We only had to add six annotations to the photo editor application, and one to the chess application, from which all components are generated automatically. After preprocessing, the tools add 574 lines of code to the photo editor application, and 332 to the chess application, which accounts for the extraction of service interfaces, the registering and lookup of services and the initialization of the OSGi framework. Each project is also split up into different subprojects for each generated bundle, and the Felix OSGi runtime and AIOLOS framework bundles are automatically added to the project. The application developer needs no knowledge about the OSGi specification and programming model, as this is handled transparently by the development tools.

9. Conclusion

In this paper we presented AIOLOS, a cyber foraging framework for Android, built on the OSGi module system and service platform. Given a number of candidate methods for offloading provided by the application developer, AIOLOS decides which of these methods should be executed remotely in view of the current context, and migrates the code at runtime to a discovered server. For each method call, AIOLOS decides whether the call is better executed remotely or locally, using a history based estimation model of the network and the processing speed of the remote server.

In order to transparently develop cyber foraging enabled applications, we provide the necessary tools that allow application developers to benefit from the AIOLOS framework only by annotating their existing Android applications.

To show the effectiveness of our approach, we annotated two Android applications for AIOLOS: Honza's Chess, an existing open source chess game, and an in-house developed photo editor application. Using these applications we show the benefits from cyber foraging by offloading to a server in the LAN network, as well as to a server deployed in the Amazon EC2 Cloud. For the specified use cases offloading offers gains of 20 to 90% in execution time depending on the mobile device capabilities, the network connectivity and the input parameters of the method calls. AIOLOS is able to estimate the remote execution time upfront with a relative precision of 10% for 80% of the method calls, using a random sample consensus algorithm. It was shown that this algorithm outperforms methods based on moving averages and least linear square fits.

As future work we plan to support stateful offloading, which introduces the need for correct data synchronization. In this aspect we can also investigate operations on data shared between multiple users in collaborative scenarios. Another research direction is to apply cyber foraging for energy saving. Here we can reuse our approach, but an accurate energy model of the mobile device is needed.

10. Acknowledgement

Tim Verbelen is funded by Ph.D grant of the Fund for Scientific Research, Flanders (FWO-V).

References

[1] Gartner Group, 2011 press releases, `http://www.gartner.com/it/page.jsp?id=1622614`.

[2] M. Butler, Android: Changing the mobile landscape, IEEE Pervasive Computing 10 (2011) 4–7.

[3] M. Satyanarayanan, Pervasive computing: Vision and challenges, IEEE Personal Communications 8 (2001) 10–17.

[4] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, I. Brandic, Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility, Future Generation Computer Systems 25 (6) (2009) 599–616.

[5] J. Flinn, S. Park, M. Satyanarayanan, Balancing performance, energy, and quality in pervasive computing, in: ICDCS '02: Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02), IEEE Computer Society, 2002, pp. 217–226.

[6] M. D. Kristensen, N. O. Bouvin, Scheduling and development support in the scavenger cyber foraging system, Pervasive and Mobile Computing 6 (6) (2010) 677–692, special Issue PerCom 2010.

[7] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, P. Bahl, Maui: making smartphones last longer with code offload, in: MobiSys '10: Proceedings of the 8th international conference on Mobile systems, applications, and services, ACM, 2010, pp. 49–62.

[8] T. Verbelen, T. Stevens, P. Simoens, F. De Turck, B. Dhoedt, Dynamic deployment and quality adaptation for mobile augmented reality applications, J. Syst. Softw. 84 (2011) 1871–1882.

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

[10] R. Kemp, N. Palmer, T. Kielmann, H. Bal, Cuckoo: a Computation Offloading Framework for Smartphones, in: ICST Conference on Mobile Computing, Applications and Services, MobiCASE, 2010.

[11] The OSGi Alliance, OSGi Service Platform, Core Specification, Release 4, Version 4.2, aQute, 2009.

[12] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, H.-I. Yang, The case for cyber foraging, in: EW 10: Proceedings of the 10th workshop on ACM SIGOPS European workshop, ACM, 2002, pp. 87–92.

[13] D. Narayanan, M. Satyanarayanan, Predictive resource management for wearable computing, in: Proceedings of the 1st international conference on Mobile systems, applications and services, MobiSys '03, ACM, 2003, pp. 113–128.

[14] R. K. Balan, D. Gergle, M. Satyanarayanan, J. Herbsleb, Simplifying cyber foraging for mobile devices, in: Proceedings of the 5th international conference on Mobile systems, applications and services, MobiSys '07, ACM, 2007, pp. 272–285.

[15] X. Gu, A. Messer, I. Greenberg, D. Milojicic, K. Nahrstedt, Adaptive offloading for pervasive computing, IEEE Pervasive Computing 3 (3) (2004) 66–73.

[16] S. Ou, K. Yang, J. Zhang, An effective offloading middleware for pervasive services on mobile devices, Pervasive and Mobile Computing 3 (4) (2007) 362–385.

[17] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, L. Yang, Accurate online power estimation and automatic battery behavior based power model generation for smartphones, in: Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, CODES/ISSS '10, ACM, 2010, pp. 105–114.

[18] I. Giurgiu, O. Riva, D. Juric, I. Krivulev, G. Alonso, Calling the cloud: enabling mobile phones as interfaces to cloud applications, in: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware, Middleware '09, Springer-Verlag New York, Inc., 2009, pp. 5:1–5:20.

[19] T. Verbelen, R. Hens, T. Stevens, F. De Turck, B. Dhoedt, Adaptive online deployment for resource constrained mobile smart clients, in: Mobile Wireless Middleware, Operating Systems, and Applications, Vol. 48, Springer Berlin Heidelberg, 2010, pp. 115–128.

[20] S. Han, S. Zhang, J. Cao, Y. Wen, Y. Zhang, A resource aware software partitioning algorithm based on mobility constraints in pervasive grid environments, Future Generation Computer Systems 24 (6) (2008) 512–529.

[21] S. Goyal, J. Carter, A lightweight secure cyber foraging infrastructure for resource-constrained devices, in: WMCSA '04: Proceedings of the Sixth IEEE Workshop on Mobile Computing Systems and Applications, IEEE Computer Society, 2004, pp. 186–195.

[22] Y.-Y. Su, J. Flinn, Slingshot: deploying stateful services in wireless hotspots, in: MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services, ACM, 2005, pp. 79–92.

[23] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, Clonecloud: Boosting mobile device applications through cloud clone execution, Tech. Rep. arXiv:1009.3088 (2010).

[24] K. Kumar, Y.-H. Lu, Cloud computing for mobile users: Can offloading computation save energy?, Computer 43 (4) (2010) 51 –56.

[25] M. A. Fischler, R. C. Bolles, Readings in computer vision: issues, problems, principles, and paradigms, Morgan Kaufmann Publishers Inc.,

1987, Ch. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography, pp. 726–740.

[26] J. S. Rellermeyer, G. Alonso, T. Roscoe, R-osgi: distributed applications through software modularization, in: Middleware '07: Proceedings of the International Conference on Middleware, Springer-Verlag New York, Inc., 2007, pp. 1–20.

[27] ProSyst mBS Mobile for Android, `http://www.prosyst.com`.

[28] Apache Felix, `http://felix.apache.org/site/index.html`.

[29] Honza's Chess, `http://honzovysachy.sourceforge.net`.

[30] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, K. R. Walker, Agile application-aware adaptation for mobility, in: Proceedings of the sixteenth ACM symposium on Operating systems principles, SOSP '97, ACM, 1997, pp. 276–287.

[31] S. Hemminger, Network Emulation with NetEm, in: Linux Conf Au, 2005.
URL `http://developer.osdl.org/shemminger/netem/LCA2005_paper.pdf`