Journal of Systems and Software
Manuscript Draft

Manuscript Number: JSS-D-11-00069

Title: Middleware for Dynamic Deployment Adaptation enabling Mobile Augmented Reality

Article Type: Special Issue: Mobile Applications

Keywords: Distributed Systems; Cyber Foraging; Deployment Optimization; Mobile Computing

Abstract: With the increasing popularity of smartphones and netbooks, more and more applications are developed for the mobile platform. Notwithstanding the recent advances in mobile hardware, most mobile devices still lack sufficient resources (e.g. CPU power, memory) to execute complex multimedia applications such as augmented reality. Application developers also have difficulties to cope with the changing device context (e.g. network connectivity, remaining battery life) and the many different hardware platforms and operating systems to run applications on. Therefore, we introduce the concept where the developer can provide different configurations of an application, each having different resource requirements and a different quality offered to the end user. The middleware framework presented in this paper will select and deploy the configuration offering the best quality possible for the current connectivity and available resources. As these change over time, the framework will dynamically adapt the configuration and deployment at runtime, enhancing the quality by offloading parts of the application when a remote server is discovered, or gracefully degrading the quality when the network connection is lost. Based on experimental results on the augmented reality use case the performance and effectiveness of our middleware has been characterized in different scenarios.

Middleware for Dynamic Deployment Adaptation enabling Mobile Augmented Reality

Tim Verbelen, Tim Stevens, Filip De Turck, Bart Dhoedt

Ghent University – IBBT, Department of Information Technology,
Gaston Crommenlaan 8 bus 201, 9050 Gent, Belgium

Abstract

With the increasing popularity of smartphones and netbooks, more and more applications are developed for the mobile platform. Notwithstanding the recent advances in mobile hardware, most mobile devices still lack sufficient resources (e.g. CPU power, memory) to execute complex multimedia applications such as augmented reality. Application developers also have difficulties to cope with the changing device context (e.g. network connectivity, remaining battery life) and the many different hardware platforms and operating systems to run applications on. Therefore, we introduce the concept where the developer can provide different configurations of an application, each having different resource requirements and a different quality offered to the end user. The middleware framework presented in this paper will select and deploy the configuration offering the best quality possible for the current connectivity and available resources. As these change over time, the framework will dynamically adapt the configuration and deployment at runtime, enhancing the quality by offloading parts of the application when a remote server is discovered, or gracefully degrading the quality when the network connection is lost. Based on experimental results on the augmented reality use case the performance and effectiveness of our middleware has been characterized in different scenarios.

Keywords: Distributed Systems, Cyber Foraging, Deployment Optimization, Mobile Computing

1. Introduction

According to a Gartner press release from 12 August 2010 [1], smartphone sales in the second quarter of 2010 totaled 61.6 million units, a 50.5 percent increase from the same period in 2009. These numbers illustrate that a significant and growing segment of the world population wants to use their handheld device in a more versatile way than the traditional mobile phone. Due to recent advances in mobile hardware, such as increased battery capacity, CPU power, display resolution, and improved network connectivity, new and more advanced services are being offered for use on handheld devices. Today, a myriad of applications can be downloaded and installed on smartphones, in a wide range of categories such as web browsers, games, social and messaging clients, location-based services, audio and video players, just to name a few. As an application developer, it is difficult to design and develop such applications specifically for the mobile market, as due to the still limited available resources (e.g. CPU power, memory) on the mobile platform mobile applications are usually less advanced in comparison with their desktop counterparts and hence cannot match the desktop experience for similar applications. It is also a burden to deal with the variety of mobile devices, different mobile platforms and the varying mobile context (e.g. connectivity, location, etc.) [2], which rises the need for run-time adaptation and recomposition as depicted in [3]

Especially for real-time demanding multimedia applications such as augmented reality (AR), the available CPU power and memory falls short to execute complex computer vision algorithms developed for the desktop. In order to run augmented reality applications on the mobile device one approach is to reduce the complexity of state-of-the-art computer vision algorithms, that lower the required CPU power but also reduce the quality compared to their desktop counterparts [4],[5]. Another approach is to split the application in a client-side and a server-side task as investigated in [6]. Object recognition is performed on a remote server, while object tracking is still done at the mobile device. The biggest drawback of outsourcing tasks to a server is that the possibly large delay between the client and server can deteriorate the usability of the application [7].

In this paper we present a hybrid approach to tackle the challenges of mobile

multimedia applications such as augmented reality. We present a middleware framework that will adapt the application depending on the mobile device and its available resources. In order to be able to adapt the application we assume that the developer provides different configurations of the application that have different resource requirements and thus offer a different quality level to the end user. For example in the object recognition scenario an object could be recognized using complex computer vision algorithms, but also by scanning the image for a product barcode. When a low latency remote server is discovered the framework will outsource parts of the application and run complex object recognition algorithms remotely. When the connection is deteriorated, the framework will try to decrease the network usage, most often resulting in more mobile processing and if necessary degrade the quality of the application by switching to barcode detection to be able to meet the real-time constraints. The end user will experience a higher quality when the complex object recognition is working, but when not enough resources are available he can still use the application using the barcode scanner.

The reminder of this paper is structured as follows. In the next section we discuss related work on code outsourcing and cyber foraging. Section 3 describes the architecture of our offloading framework and section 4 presents our method to calculate the optimal deployment of the application. In section 5 a detailed use case is described which is used to evaluate our framework in section 6 and finally section 7 concludes this paper.

2. Related work

In this section an overview is given of related work done in automatic partitioning and remote execution of software. Early software partitioners such as JavaParty [8], Doorastha [9] and AdJava [10] use an additional preprocessing step before compiling to insert remote invocation code and expect the programmer to insert special keywords indicating parts of the software to be run remotely. However, these methods fully rely on the application programmer and the source code is needed to partition the program.

This problem is addressed by using binary rewriting to introduce remote execution of parts of the software. This technique is used by Addistant [11] and J-Orchestra [12] that impact the Java bytecodes to create a distributed

application. Similarly, Coign [13] distributes Microsoft COM objects and MAUI [14] rewrites programs written for the Microsoft .NET Common Language Runtime. In order to decide which parts to offload, an offline profiling phase is used to calculate the optimal partitioning.

With the emergence of pervasive computing, the idea of cyber foraging was introduced by Satyanarayanan [15]. The idea is to use available resources in the neighbourhood of a mobile device to outsource parts of the application and utilize the available resources to increase the performance on the device. The earliest cyber foraging systems such as Spectra [16] and Chroma [17] introduce a tactics based scheduler to decide which methods to outsource. However, both systems rely on pre-installed remote procedure calls, implying that all remoting has to be programmed by the application developer.

A better approach is to make use of mobile code, where the system is able to outsource parts of the software at runtime, with minimal interference from the developer. Gu et al. [18] present an adaptive offloading framework where Java classes can be placed remotely. A fuzzy control model is used to offload classes at runtime and an adapted MINCUT heuristic is used to minimize interactions between partitions. Ou et al. also use class outsourcing, proposing a (k+1) partitioning algorithm to outsource k parts on k remote servers [19]. Han et al. present a flow-based algorithm to partition software which is evaluated by simulation [20].

The Scavenger cyber foraging [21] system uses a scheduler using adaptive history-based profiling in order to outsource Python methods, where candidate methods for outsourcing have to be annotated by the application developer. Zhang et al. [22] propose a mobile code framework where platform independent software components – called weblets – can be outsourced to the cloud based on a bayesian learning scheduler.

Lastly, recent advances in cloud computing have led to the use of virtual machines in the cloud as surrogates [23]. The biggest problem in using the cloud for cyber foraging is presented by the typically high latency of the WAN link compared to the WLAN connection. To cope with this bottleneck Su et al. proposed Slingshot [24], where the VMs are co-located with the wireless access point. Satyanarayanan et al. [25] also placed the surrogates, called cloudlets, in the physical proximity of the mobile user. Chun et al. [26] uses

virtualization technology 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 a framework for adaptive deployment that adopts the cyber foraging concept to offload parts of the software to nearby discovered servers. Instead of focusing on few well defined tasks as candidates for outsourcing such as image filtering [21], [22] or speech recognition [23], we focus on real-time multimedia applications such as augmented reality where multiple components are candidate for outsourcing. We take the same assumption as Satyanarayanan et al. [25], which implies that the remote processing power is in the vicinity of the user, since high latencies on a WAN link make it useless to offload for real-time applications. In order to target a wide range of mobile platforms and to be able to cope with the changing context, an application developer can provide high quality and low quality versions of components in the application. We present a method to calculate the optimal deployment given the current connectivity and available resources that maximizes the overall quality of the application, rather than focusing solely on minimizing memory [18], CPU [19] or energy requirements [20]. Our middleware is able to dynamically adapt the application when the device context changes, i.e. offload parts to a remote server when one is discovered, or gracefully degrade to a lower quality version of the application when the network connection is lost, taking into account the real-time constraints.

3. Offloading middleware overview

3.1. Requirements

To build such an offloading framework, different requirements have to be filled in. The application has to be able to adapt to the ever changing available resources, which must happen at run-time since it is impossible to predict all contexts the application has to operate in. This means the framework has to be able to sense the context to gather information about all available resources, which is done by monitoring all known resources on the one hand, and by discovering possible remote resources on the other hand.

In order to offload parts of the software to a remote server a mobile code approach is adopted. This means units of deployment have to be defined that

can migrate from one machine to another, a remote procedure call technique has to be used to call these offloaded components, while keeping this as transparent as possible to the application programmer.

Last but not least an algorithm is needed to decide when to redeploy and which deployment is best given the current device context. Given the fact that one application can consist of different versions of different quality levels, the goal is to deploy the best quality version that is feasible given the available resources. The details of this algorithm are described in Section 4.

3.2. Offloading framework

This section presents the different components of our offloading framework as illustrated on Figure 1. Our middleware is built upon OSGi [27], a module system and service platform that enables runtime deployment of application components – called bundles – implemented in Java. The portability of Java enables our framework to migrate and execute code on different hardware platform and architectures.

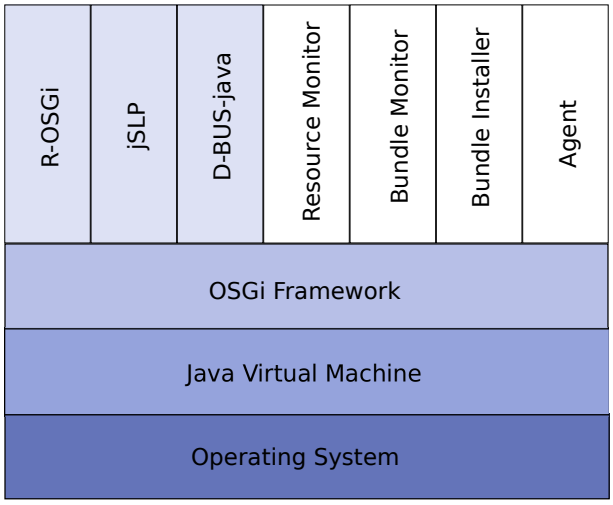


Figure 1: Overview of the offloading middleware. The Agent fetches the monitor information from the monitor bundles and calculates the best deployment. The Bundle Installer distributes the components to the discovered servers. Our framework is built upon the OSGi framework in Java and uses R-OSGi for remote execution, jSLP for server discovery and DBUS as an interface to the operating system.

The OSGi bundle concept gives us a unit of deployment, which can be migrated to other machines, while being transparent for the application developer: by developing the application as OSGi bundles the framework will be able to partition it and outsource parts of it without the need of special code or annotations.

On top of the OSGi framework, the following three components provide different functionalities:

R-OSGi extends the functionality of OSGi for distributed systems [28] and provides remote invocation of methods exposed by the different OSGi bundles. By creating a local proxy of a bundle running on a remote server application functionality can be outsourced transparently for the application.

jSLP is a pure Java implementation of the Service Location Protocol (SLP) as specified in RFC 2608 [29]. It has a small footprint which makes it feasible for embedded devices and is used for the discovery of remote servers in our framework.

Java D-Bus offers access to the D-Bus message bus system, which is used for inter-process communication between different desktop applications. D-Bus is commonly used on Linux-based systems and enables us to integrate applications that use our middleware in the operating system, for example start-up through shortcuts.

The last four bundles make up our offloading middleware:

Resource Monitor collects system wide monitoring information such as the used and available CPU time and bandwidth.

Bundle Monitor fetches monitor information on a more fine-grained level. This bundle monitors the CPU time used for each bundle and captures the communication between different bundles.

Bundle Installer is responsible for the installation and migration of application bundles. When a bundle is migrated the Bundle Installer will send the code to the destined machine, start it up there and set up all proxies to enable remote execution through R-OSGi.

Agent is the core of our offloading framework. The Agent keeps track of the remote servers discovered by jSLP and periodically checks the monitoring information collected by the Resource Monitor and the Bundle Monitor to decide if a redeployment is needed. Using the current available bandwidth and CPU resources the best deployment is calculated – on a mobile device this calculation is usually outsourced to the server side – and the Bundle Installer is instructed to migrate the necessary bundles. The algorithm for determining the best deployment is presented in the next section.

4. Optimal deployment calculation

We assume that the application developer provides all software bundles that make up an application. An application can consist of several configurations where different configurations can implement the application on a different quality level. For example in a location based application, there can be a bundle to estimate the location by using the GPS sensor or by using GSM localization using the roaming signal to the antenna tower. This can result in two different configurations where the framework will choose the most appropriate configuration offering the best quality possible in the current device context. For each configuration also different deployments are possible, as bundles can be migrated to remote servers if available. An example application with two configurations is shown on Figure 2.

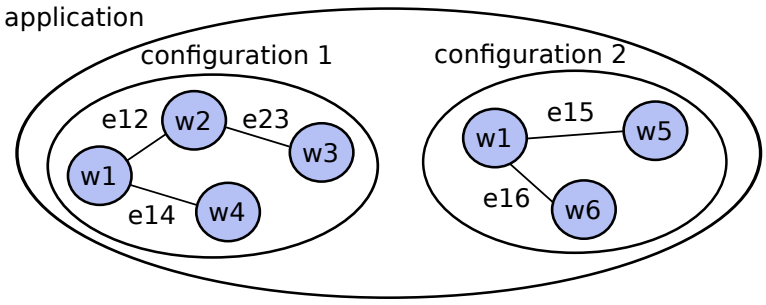


Figure 2: The application developer provides an application consisting of 6 bundles. These can be used in two configurations that each offer a different quality level. Each bundle has an associated resource cost w_i and interacting bundles have a communication cost e_{ij}. The framework will decide at runtime which configuration to deploy and which bundles should be offloaded.

More formally an application can be represented as a set of possible graphs $G = (V, E)$, each representing a single configuration, where V is a set of N vertices and E is a set of edges connecting these vertices. The vertices represent the software bundles in a configuration and the edges represent communication between those bundles. We associate a cost w_i with each vertex v_i indicating the amount of resources (i.e. CPU power) this component needs. $C = (c_{ij})$ is the adjacency matrix of G, i.e. if there exists an edge between v_i and v_j then c_{ij} equals the weight of this edge, otherwise $c_{ij} = 0$. The edge weights represent the cost of communication (i.e. bandwidth) between different software components. The weights of the vertices and edges can be found by profiling the different configurations of the application. Each software bundle can be deployed on the mobile device or on a discovered server with respectively M_0 and M_1 maximum available resources. The wireless link has a maximum link capacity denoted by L.

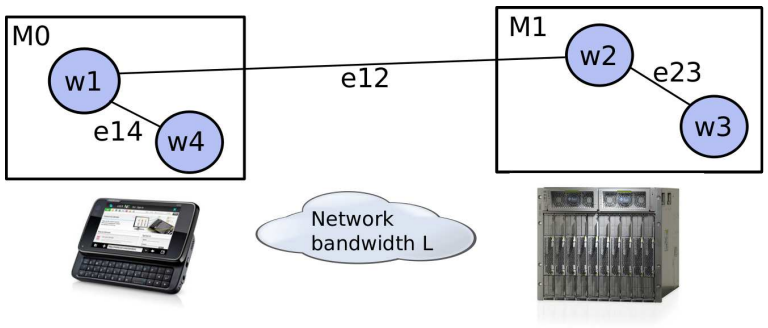


Figure 3: The best configuration is the one that matches with the available resources on the mobile device (M0) and a possibly discovered server (M1) and the available bandwidth (L) while maximizing the perceived quality. This configuration is then deployed on the mobile device and the server.

The goal now is to find the configuration where all bundles can be deployed on either the mobile device or the discovered server, taking in account their maximum available resources and the wireless link capacity, and which offers the best user experience, or thus maximizing the quality of the application.

We assume here that the quality of software bundle v_i increases with its

weight w_i meaning that the software bundle does more useful work, as it is useless to sacrifice more CPU power for lower quality. The best configuration therefore maximizes the total node weights, i.e.

$$\max_{configurations} \sum_i w_i \tag{1}$$

However, due to the constrained available resources and limited bandwidth, it is not always possible to deploy this configuration. Therefore we calculate a possible deployment for each configuration by solving the following ILP problem for one specific configuration and its associated graph G. Let X_{im} be the decision variable that is equal to 1 if software bundle v_i is assigned to machine $m \in 0, 1$ and 0 otherwise.

To end with a possible deployment following constraints are imposed. First of all the sum of the weights of all bundles deployed on the device or the server cannot exceed the maximum allowed weight.

$$\forall m : \sum_i X_{im} \times w_i \leq M_m \tag{2}$$

Next, in order to have a fully functional application, each bundle of the configuration has to be deployed on either the device or the server.

$$\forall i : \sum_m X_{im} = 1 \tag{3}$$

Lastly the bandwidth needed should be less than the available bandwidth L. Therefore the bandwidth usage for the configuration is minimized as the objective function of our ILP problem and afterwards is checked if the resulting bandwidth is less than L.

Let h_{ij} take the value 1 when v_i and v_j are deployed on a different machine, 0 otherwise. Then the objective function becomes:

$$\min \frac{1}{2} \sum_i \sum_j h_{ij} \times c_{ij} \tag{4}$$

Variables h_{ij} can be expressed as a function of decision variables X_{im} as follows:

$$\forall i, j : h_{ij} = 1 - \sum_{m} X_{im} \times X_{jm}, \tag{5}$$

Indeed, if v_i and v_j are deployed on the same machine M, all X_{im} and X_{jm} will be zero except for $m = M$ where they are both 1, making the resulting sum equal to 1. When they are deployed on different machines, for each m at least one of the two variables X_{im} or X_{jm} will equal 0. Replacing 5 in equation 4 gives a formal description that can be solved with a solver that can handle quadratic objective functions (e.g. CPLEX [30]). It can also be converted to a true Integer Linear Programming (ILP) problem by stating that:

$$\forall i, j : h_{ij} = 1 - \sum_{m} h_{ijm}, \tag{6}$$

Equation 6 then introduces extra decision variables h_{ijm} subjected to the following constraints:

$$\forall i, j, m : \begin{cases} h_{ijm} \leq X_{im} \\ h_{ijm} \leq X_{jm} \\ h_{ijm} \geq X_{im} + X_{jm} - 1 \end{cases} \tag{7}$$

The final algorithm to calculate the best deployment is illustrated on Figure 4. The configurations are processed in order of decreasing total weight. When no remote server is discovered, the configuration is chosen with maximum total node weight that fits on the client device or thus where $\sum_i w_i \leq M_0$. If no such solution exists the configuration is chosen with minimal total node weight. When a remote server is available, the ILP problem is solved and if a solution exists with bandwidth smaller of equal to the available bandwidth L this result is returned.

Note that the ILP problem – which is computationally expensive – only has to be solved when a remote server is available. This calculation can thus be outsourced and executed at the server side which can solve this problem in less than a second for graphs of 10 to 20 vertices. For larger graphs a set of deployments could also be calculated upfront for a wide range of parameters.

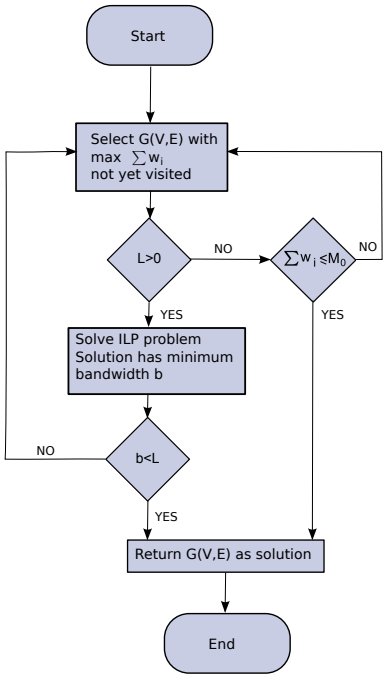


Figure 4: Flowchart of the algorithm to calculate the best deployment out of different possible configurations.

5. Augmented reality use case

As use case for our offloading framework, we focus on augmented reality applications for two main reasons. Firstly, it is much more challenging to identify the best deployment and identify good candidates for outsourcing in an augmented reality application due to its real-time character, as opposed to non real-time applications such as a photo gallery application where the execution of an image filter is almost always a good task to outsource. Secondly the complexity of augmented reality and the various computer vision algorithms used make it a mere necessity to outsource parts of the application in order to be able to run the most advanced techniques on the mobile device.

We present an augmented reality shopping assistant that offers the user extra information on the product(s) in view in the form of an overlay. Because

we want the application to work in different circumstances and on various devices with different hardware, we provide different configurations to detect products: one way is to scan the product by its barcode, another way is by visual object recognition. When few resources are available the user has to find the barcode and hold it into view. When a remote server is available more complex object recognition algorithms can be used and the object is recognized immediately, thus offering a better quality to the end user at the cost of more required resources.

The general architecture and different components of this application are illustrated in the sequence diagram on Figure 5. The Video bundle has a separate thread to continuously capture frames and show them on the screen. Another thread is run by the Analyzer bundle that will fetch the last captured frame and analyze it for possible objects. The Analyzer passes the frame to the Detector that can identify possible interesting feature points or regions. Next the Recognizer/Decoder bundle will try to decode a barcode from the detected region or match it to a known set of feature points associated to a certain object. When an object is found its ID is passed to the ContentProvider bundle that fetches the associated content that is then displayed on top of the video.

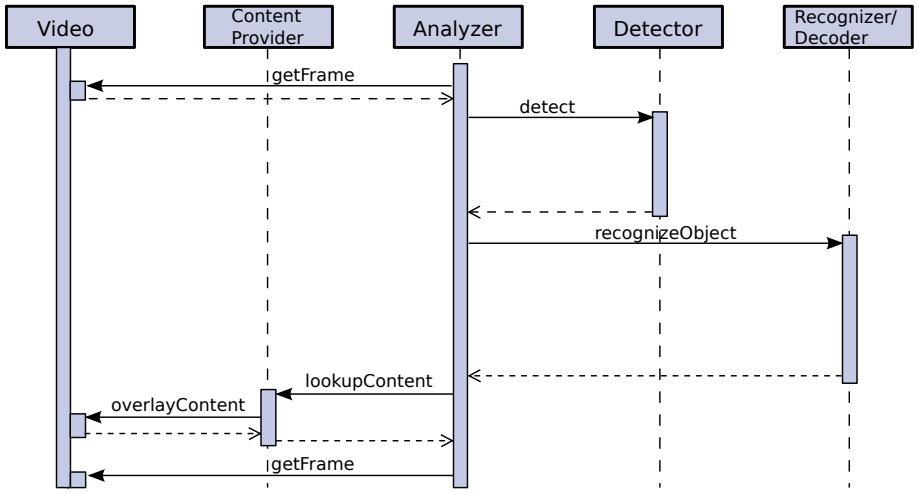


Figure 5: Sequence diagram of the augmented reality shopping assistant.

Of each component type one or more implementations are developed.

Video – Because the Video bundle is the same for all configurations, only one implementation is provided. This bundle also has to access the device camera and can not be outsourced. The video frames are captured in a native thread by use of the Java Native Interface (JNI) and rendered on screen with OpenGL ES. All other components are pure Java implementations in order to be able to migrate.

Content Provider – The Content Provider bundle simply looks up product information by product ID. Currently only one provider is implemented.

Analyzer – The Analyzer bundle contains the main application loop and orchestrates between the different bundles. There are two versions of this bundle differentiated by the way they fetch the captured frames: one fetches the image at full 640x480 resolution, while the other one subsamples the image to a 320x240 resolution.

Detector – There are two types of detector bundles depending on the method of object identification. The first type is a Barcode Detector that returns a region that possibly contains a barcode. We implemented two versions : one detector analyzes the full image for image regions with many vertical edges using Sobel filtering that possibly represent barcodes, a more simple implementation only looks at the centre image patch. The second type detector will detect image features for object recognition. Again we implemented two versions: one generates SURF feature descriptors as described in [31], which results in scale invariant and rotation invariant feature descriptors, and a faster implementation combines FAST corner detection with SURF-like feature descriptors inspired by [5], which gives up the scale invariance for more speed.

Recognizer/Decoder – Depending on the type of detector bundle a suitable implementation of the Recognizer/Decoder bundle is deployed. The Feature Recognizer matches the features from the image to a set of known features of objects to recognize. We implemented a simple matching algorithm based on cross-correlation matching that lets us differentiate a few objects, but for large product databases a more complex algorithm should be used like the one in [6]. A Barcode Decoder bundle will try to decode a possible barcode in the region detected by

Table 1: Different configurations for the AR use case.

	config 1	config 2	config 3	config 4	config 5
Video	x	x	x	x	x
ContentProvider	x	x	x	x	x
Analyzer					
- Full resolution	x		x		x
- Half resolution		x		x	
Detector					
- Centre Barcode Detector	x				
- Sobel Barcode Detector		x			
- FAST Feature Detector			x		
- SURF Feature Detector				x	x
Recognizer/Decoder					
- Barcode Decoder	x	x			
- Feature Recognizer			x	x	x

one of the Barcode Detectors, using the ZXing barcode image processing library [32].

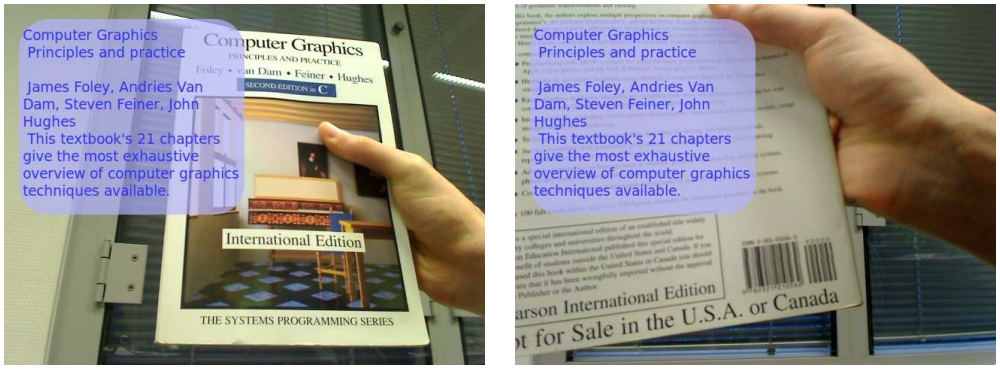
By combining the different implementations of the Analyzer, Detector and Recognizer/Decoder bundles the following five configurations can be distilled as shown in Table 1.

1. The first one combines the Centre Barcode Detector and Barcode Decoder with the 320x240 Analyzer. This configuration will need the least CPU power, but will also offer the least quality to the end user: he/she will always have to make sure the barcode of the product is in the centre of the view to get the information displayed.
2. A second configuration uses the Sobel Barcode Detector that will scan the whole image together with the 320x240 Analyzer and Barcode Decoder. Now the information will be displayed when the barcode is in view, but it does not necessarily need to be in the centre.
3. The third configuration will combine the 640x480 Analyzer with the FAST Feature Detector and the Feature Recognizer. Because the FAST feature detection the detection phase can be executed on the device and only the detected feature data has to be transmitted to the server which

can be beneficial when not much bandwidth is available. However, due to the lack of scale invariance the object can only be detected when it has the same size in the image as in the reference image.

4. The fourth configuration uses the 320x240 Analyzer with the SURF Feature Detector and the Feature Recognizer. Because the SURF feature detection is rather slow, it will probably be outsourced and thus the whole 320x240 frame will have to be sent to the server.
5. The final configuration uses the 640x480 Analyzer with the SURF Feature Detector and the Feature Recognizer. By using the full resolution image objects can be identified from a bigger distance, but at the cost of more processing power and most likely more bandwidth when the detector component is outsourced.

Figure 6 shows two screenshots of different configurations of the application. In the left screenshot configuration 5 is deployed and the book is recognized by its visual features. In the right screenshot the barcode is used to identify the object.



(a) Config 5 (b) Config 2

Figure 6: The left screenshot shows the high quality configuration where the book is visually recognized. In the right screenshot the barcode has to be in view to recognize the object.

6. Experimental validation

In this section we validate our offloading framework using the use case presented in Section 5 and a number of relevant scenarios. The first scenario is adaptation on the device locally, when the available CPU changes (e.g. when another application starts up in the background). A second scenario shows how our framework is able to discover a remote server on the network and outsources some components to this discovered server. The third scenarios introduces changes in the available bandwidth on the network and illustrates how the application is adapted accordingly. Lastly we show how the framework detects network failures and tries to recover from this situation.

6.1. Set-up

We first describe the hardware and software set-up used for our experiments. As mobile device we used a Nokia N900 smartphone equipped with a 600 MHz ARM Cortex A8 processor and 256 MB RAM. This device runs Maemo 5 Linux on which we run a Sun Java SE for Embedded 6 JVM [33] and a Felix OSGi instance [34], adapted to be able to monitor the bundles as described in [35]. It has a camera capable of video recording at a resolution of 640x480. The server machine is equipped with an Intel Core 2 DUO P8400 CPU clocked at 2.26GHz and runs Ubuntu Linux.

The mobile device is connected to the server with a USB cable and they communicate using Ethernet over USB. The bandwidth on the link can be controlled using the Linux traffic control (tc) tool. The introduced bandwidth value is also fed into our monitoring bundle since algorithms for accurate bandwidth estimation using probe traffic is out of scope of this research [36].

Finally we have to set up the different parameters of our offloading algorithm in order to correctly deploy our augmented reality application. As shown in the sequence diagram on Figure 5, the whole application consists of a repetitive loop of fetching a frame, analyzing it, detecting an object and augmenting the view. This repetitive structure is typical in multimedia applications that involve video processing. Using the Bundle Monitor we collect profile data for each configuration, where we collect for one loop the time (in ms) spent in each component (vertex weight w_i) and the data communicated

Table 2: Measurements of the CPU time spent (in ms) in the components for each application loop for the different configurations on the N900.

	config 1	config 2	config 3	config 4	config 5
Video	40	20	40	20	40
ContentProvider	40	40	40	40	40
Analyzer					
- Full resolution	10		10		10
- Half resolution		10		10	
Detector					
- Centre Barcode Detector	50				
- Sobel Barcode Detector		150			
- FAST Feature Detector			220		
- SURF Feature Detector				2390	8990
Recognizer/Decoder					
- Barcode Decoder	20	20			
- Feature Recognizer			500	820	2810

(in bits) at each method call (edge weight c_{ij}). The experimental values are shown in Table 2.

The remaining parameters to set are the maximum client and server weights M_0 and M_1 and the available bandwidth L. In order to calculate a meaningful deployment one has to impose a deadline on how long one single loop can take. For example for our augmented reality application we want to process a frame in at most 500 ms, which is an intuitively defined threshold that means one should recognize an object within half a second. The available client weight is then set to the deadline (500 ms) multiplied by the available percentage of CPU power. For example, when there is 90% CPU available, the application has to execute the loop within the 450 ms processor time available.

Of course, since the CPU of the server is more powerful than the CPU of the mobile device, the instructions executed in 1 ms on the client device will be executed much faster on the server. Therefore to determine the server weight M_1 one has to multiply by a scale factor, which can be found by comparing execution times of the same bundle on the client device and the server device (e.g. in our set-up the server executes about 30 times faster than the mobile

device). Thus suppose there is 50% CPU time available at the server, M_1 is given by 500 ms $\times 50\% \times 30 = 7500$ ms. Lastly the available bandwidth which is expressed as bits per second has to be rescaled taking into account the deadline: in our case it has to be rescaled to bits per half a second.

6.2. Results

6.2.1. Scenario 1: CPU adaptation

In the first scenario the user is running the shopping assistant application without connection to any remote resources. However, during shopping, the user wants to listen to some background music and starts up the FM radio player. Because the available CPU has to be shared between both applications, the framework will adapt the application to reduce the CPU usage.

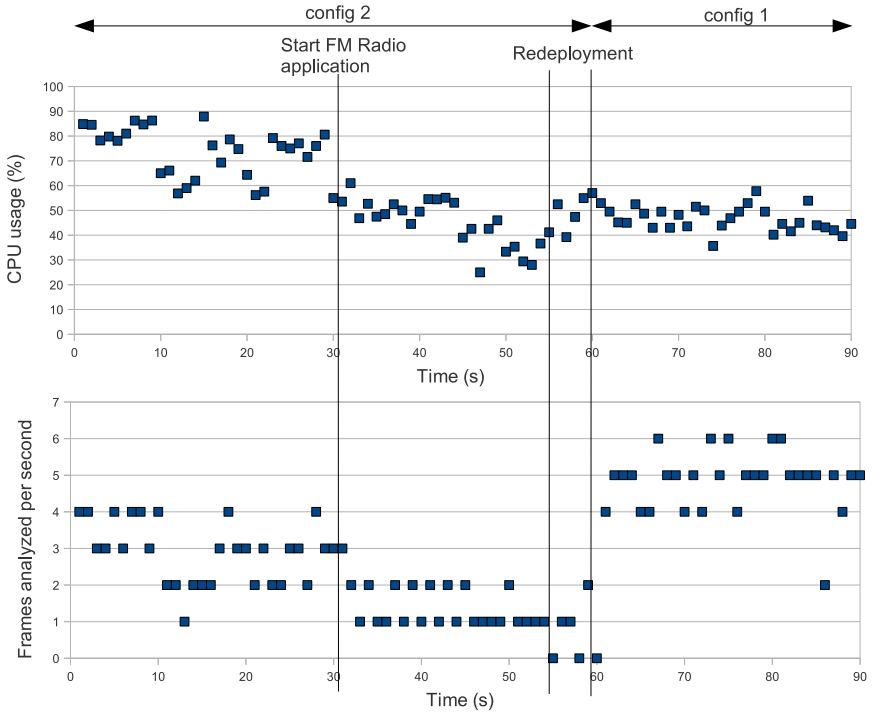


Figure 7: Scenario 1. The upper graph shows the CPU usage on the mobile device, the lower graph shows the frames analyzed per second of the shopping assistant application. When another application is started, a lower quality configuration is deployed.

Figure 7 shows what happens in this scenario: the upper graph presents the CPU usage of the mobile device running the shopping assistant application, the lower graph shows the frames analyzed per second by the shopping assistant application. At first configuration 2 is deployed: frames are analyzed for detecting barcodes using sobel filtering and when a barcode is decoded the product information is shown on the screen. This happens at a rate of 3 to 4 analyzed frames per second using all CPU power.

At $t = 30s$ the FM Radio player application is started and less CPU power is available for our application, which lead to less frames analyzed per second. The Agent performs a check at $t = 55s$ and detects less CPU power is available by calling the Resource Monitor. After recalculation the Agent instructs the Bundle Installer to change to configuration 1, where only the centre of the frame is analyzed. After redeploying the application again reaches 5 to 6 analyzed frames per second, but at the cost of quality to the user: he will have to capture the barcode in the centre of his view before the object is identified.

6.2.2. Scenario 2: Server discovery

Now the user enters a supermarket that wants to enable the customers to use more advanced mobile applications and has a set up a local WLAN network with a server where components can be outsourced to. On entering the supermarket the user's mobile device will discover the server and reconfigure the application. The user has good connectivity with the WLAN network and the available bandwidth is 10 Mbit/s.

At first configuration 2 is deployed because no remote server is found and no other applications are running on the device. When the jSLP bundle discovers a server, the Agent is notified and R-OSGi is configured to create the needed endpoints for remote calls. The Agent then gathers all resource monitor information of both the client and the discovered server and a new deployment is calculated. Since there is high bandwidth available the Bundle Installer is instructed to start configuration 5 and outsource all components except the Video bundle. This means that all full resolution images are sent to the server for analysis. Figure 8 shows this process: at $t = 36s$ the remote server is discovered and the client is configured to be able to

make remote calls which takes about 10 seconds. Then the components consisting of configuration 5 are started and outsourced, which is finished at $t = 70s$. One notices that during configuration of the server and the redeployment the number of frames analyzed per second drops, since quite some CPU goes to the redeployment and the responsiveness of the application will diminish. After redeployment, the user will have a higher quality version of the application running, and objects will be identified just by looking at them, rather than searching for a barcode and scanning it. The bandwidth usage is then around 7 MBit/s which is the bandwidth needed to send 3 frames per second (1 frame $= 640 \times 480 \times 8$ bits).

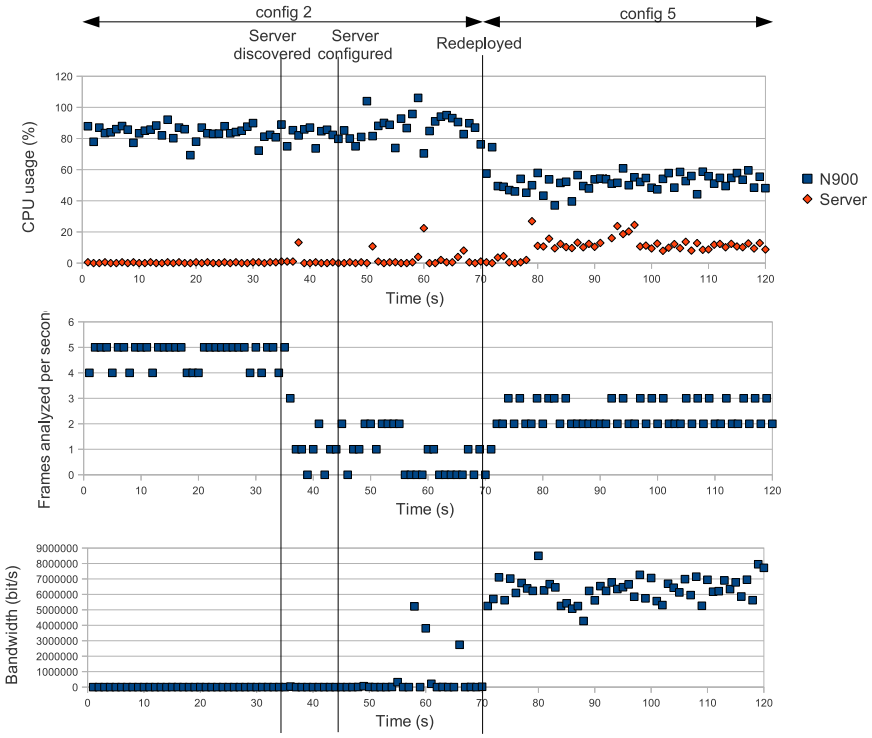


Figure 8: Scenario 2. The upper graph shows the CPU usage on the mobile device and the server, the middle graph shows the frames analyzed per second of the shopping assistant application and the lower graph shows the bandwidth usage. When a remote server is discovered, a higher quality configuration is deployed and components are outsourced.

6.2.3. Scenario 3: Bandwidth adaptation

In the third scenario the available bandwidth changes and the framework will adapt the application accordingly. At the start there is an available bandwidth of 10 Mbit/s, but it is first lowered to 3 Mbit/s, and later even to 750 Kbit/s. The framework will deploy respectively configuration 4 (sending lower resolution frames) and configuration 3 (keep feature detection local and only send detected features to the server) to lower the bandwidth needed by the application as shown on Figure 9.

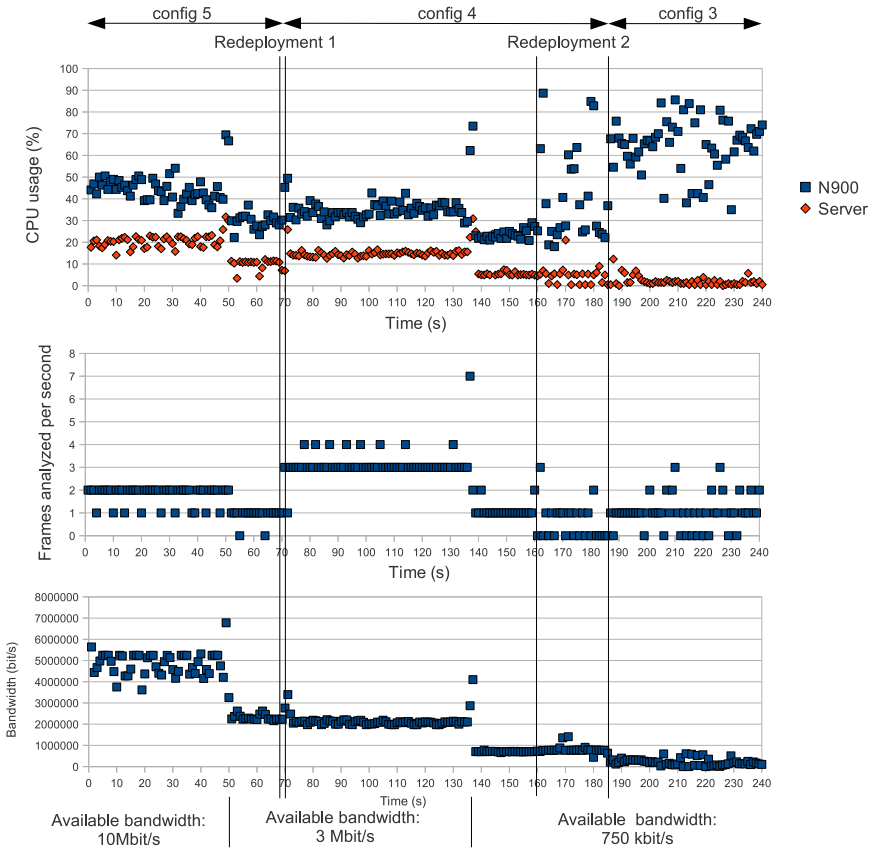


Figure 9: Scenario 3. The upper graph shows the CPU usage on the mobile device and the server, the middle graph shows the frames analyzed per second of the shopping assistant application and the lower graph shows the bandwidth usage. As the available bandwidth is lowered, the framework deploys lower quality configurations that use less bandwidth.

Initially a hight bandwidth network is available and configuration 5 that provides the highest quality is deployed. At $t = 50s$ the available bandwidth is lowered from 10 to 3 Mbit/s. Due to the lower bandwidth, the number of analyzed frames drops from 1-2 to 0-1 and the framework decides to redeploy. This redeployment takes little time since only one bundle has to be changed and both on the device and the server there is CPU available to start and outsource this bundle.

At $t = 135s$ the available bandwidth is even more lowered to 750 Kbit/s. Again the analyzed frames per second are lowered and the framework decides again to redeploy. Now more bundles have to be started and little bandwidth is available to outsource bundles, leading to a redeployment time of 25 seconds. Again during redeployment the number of analyzed frames per seconds diminishes.

6.2.4. Scenario 4: Network failure recovery

When the connection to the server is lost due to a network failure, the framework will detect the remote server is no longer available and recover from this failure by starting configuration 2, which can run locally on the device.

Figure 10 shows this scenario. Initially configuration 5 is deployed using the discovered server on a 10 Mbit/s network and at $t = 50s$ the server is stopped causing no more frames to be analyzed. At $t = 75s$ the framework detects the failure and switches back to the local configuration 2.

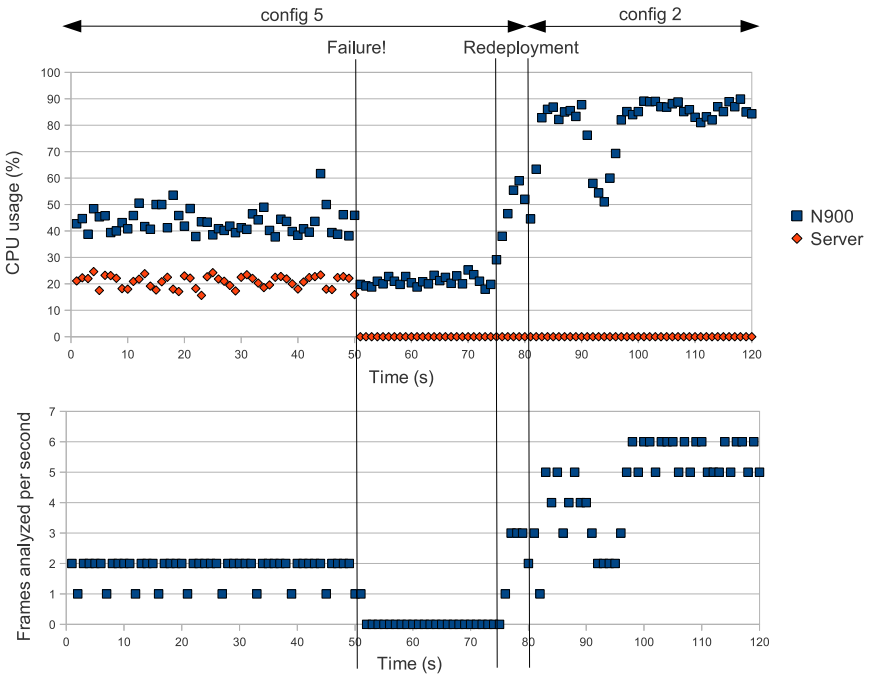


Figure 10: Scenario 4. The upper graph shows the CPU usage on the mobile device and the server, the lower graph shows the frames analyzed per second of the shopping assistant application. When there is a failure of the server or the network, the framework recovers from the failure and switches back to a degraded version of the application.

6.3. Discussion

In four scenarios, we have demonstrated the validity of our framework, the suitability to develop applications in different quality configurations and how our framework switches between those configurations. The only issue that can be experienced by the user is a drop in performance during the redeployment phase, especially in the case when there is not enough CPU power left on the device (Redeployment in Scenario 2) to start the bundles, or when there is little bandwidth available to outsource the bundles (Redeployment 2 in Scenario 3). However, in this experiment all bundles are started 'cold', meaning they are only started when they are needed. The redeployment time could be further lowered by caching selected bundles of all configurations at both the device and the server, ready to be used when a redeployment occurs.

Because the bandwidth limit is imposed using the Linux Traffic Control (tc) tool, the traffic is actually shaped when the available bandwidth limit is exceeded. In a real network setting, this is not the case and trying to send more packets in the network would lead to packet loss, which will deteriorate the application quality even more and make it more important to redeploy in order to reduce bandwidth usage.

7. Conclusions and future work

In this paper we presented a mobile middleware for adaptive deployment of multimedia applications, specifically augmented reality. Using the use case of an augmented reality shopping assistant, we introduce the concept of multiple configurations of an application offering different qualities. We presented an algorithm for calculating the deployment offering the best quality possible given the current connectivity and available resources. Our framework is built upon the OSGi framework and is able to switch configurations and outsource components at runtime. When a remote server is discovered a higher quality configuration is deployed as more load can be outsourced. When the connection is lost, the application gracefully degrades to a lower quality configuration. We presented experimental results of different scenarios with an implementation of our augmented reality use case that illustrate the performance and effectiveness of our approach.

Important points for future work are to lower the redeployment time by studying caching strategies for bundles and to evaluate our middleware in other scenarios.

8. Acknowledgement

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

References

[1] Gartner Group, "2010 press releases", `http://www.ilog.com/products/cplex/`.

[2] A. Fortier, G. Rossi, S. E. Gordillo, C. Challiol, Dealing with variability in context-aware mobile software, Journal of Systems and Software 83 (6) (2010) 915–936.

[3] N. Gui, V. D. Florio, H. Sun, C. Blondia, Toward architecture-based context-aware deployment and adaptation, Journal of Systems and Software 84 (2) (2011) 185–197.

[4] G. Klein, D. Murray, Parallel tracking and mapping on a camera phone, in: ISMAR '09: Proceedings of the 8th IEEE International Symposium on Mixed and Augmented Reality, IEEE Computer Society, Washington, DC, USA, 2009, pp. 83–86.

[5] D. Wagner, G. Reitmayr, A. Mulloni, T. Drummond, D. Schmalstieg, Real-time detection and tracking for augmented reality on mobile phones, IEEE Transactions on Visualization and Computer Graphics 16 (2010) 355–368.

[6] S. Gammeter, A. Gassmann, L. Bossard, T. Quack, L. Van Gool, Server-side object recognition and client-side object tracking for mobile augmented reality, in: IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), 2010, pp. 1–8.

[7] J. J. Hull, X. Liu, B. Erol, J. Graham, J. Moraleda, Mobile image recognition: architectures and tradeoffs, in: Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications, HotMobile '10, ACM, New York, NY, USA, 2010, pp. 84–88.

[8] M. Philippsen, M. Zenger, Javaparty – transparent remote objects in java, Concurrency: Practice and Experience 9 (11) (1997) 1225–1242.

[9] M. Dahm, Doorastha – a step towards distribution transparency, in: JIT, 2000.

[10] M. M. Fuad, M. J. Oudshoorn, AdJava: automatic distribution of java applications, in: ACSC '02: Proceedings of the twenty-fifth Australasian conference on Computer science, Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2002, pp. 65–75.

[11] M. Tatsubori, T. Sasaki, S. Chiba, K. Itano, A bytecode translator for distributed execution of "legacy" java software, in: Object-Oriented Programming, Springer-Verlag, 2001, pp. 236–255.

[12] E. Tilevich, Y. Smaragdakis, J-Orchestra: Enhancing java programs with distribution capabilities, ACM Transactions on Software Engineering and Methodology 19 (1) (2009) 1–40.

[13] G. C. Hunt, M. L. Scott, The coign automatic distributed partitioning system, in: OSDI '99: Proceedings of the third symposium on Operating systems design and implementation, USENIX Association, Berkeley, CA, USA, 1999, pp. 187–200.

[14] 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, New York, NY, USA, 2010, pp. 49–62.

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

[16] 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, Washington, DC, USA, 2002, pp. 217–226.

[17] 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, New York, NY, USA, 2002, pp. 87–92.

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

[19] 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.

[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] 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.

[22] X. Zhang, S. Jeong, A. Kunjithapatham, S. Gibbs, Towards an elastic application model for augmenting computing capabilities of mobile platforms, in: Mobile Wireless Middleware, Operating Systems, and Applications, Vol. 48, Springer Berlin Heidelberg, 2010, pp. 161–174.

[23] 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, Washington, DC, USA, 2004, pp. 186–195.

[24] 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, New York, NY, USA, 2005, pp. 79–92.

[25] 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.

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

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

[28] 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., New York, NY, USA, 2007, pp. 1–20.

[29] E. Guttman, C. Perkins, J. Veizades, M. Day, Service location protocol, version 2 (1999).

[30] IBM ILOG CPLEX, http://www.ilog.com/products/cplex/.

[31] H. Bay, A. Ess, T. Tuytelaars, L. Van Gool, Speeded-up robust features (surf), Computer Vision and Image Understanding 110 (2008) 346–359.

[32] ZXing, http://code.google.com/p/zxing.

[33] Sun Java SE for Embedded 6, http://java.sun.com/javase/embedded/index.jsp.

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

[35] T. Verbelen, R. Hens, T. Stevens, F. 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.

[36] C. D. Guerrero, M. A. Labrador, On the applicability of available bandwidth estimation techniques and tools, Computer Communications 33 (1) (2010) 11 – 22.