Journal of Network and Computer Applications 41 (2014) 206–216

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Adaptive deployment and configuration for mobile augmented reality in the cloudlet

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

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

ARTICLE INFO

Article history:
Received 5 April 2013
Received in revised form
26 November 2013
Accepted 3 December 2013
Available online 28 December 2013

Keywords:
Cloudlet
Mobile computing
Augmented reality
OSGi

ABSTRACT

Despite recent advances in mobile hardware, most mobile devices still fall short to execute complex multimedia applications with real-time requirements such as augmented reality (AR). Because offloading the application to the cloud is not always an option due to the high and often unpredictable WAN latencies, the concept of cloudlets has been introduced: nearby infrastructure offering virtual machines for remote execution.

In this paper we present a cloudlet platform, providing two important contributions. First, the platform allows cloudlets to be formed in a dynamic way, including (fixed) virtualized infrastructure co-located with the wireless access point, as well as any device in the LAN network supporting the platform. The approach can also be extended towards the cloud, facilitating distribution of applications over three tiers (i.e., the device, the cloudlet and the cloud). Second, instead of moving a complete virtual machine to the cloudlet, we propose a more fine-grained approach, by managing and deploying applications on the component level. Application components are declared by the developer, together with their real-time constraints and configuration parameters. In order to meet these constraints and to optimize the user experience, the platform distributes these components among the cloudlet at runtime while also dynamically configuring parameters.

An OSGi-based prototype implementation on the Android platform is highlighted and evaluated using a mobile AR use case, showing the need for a component-based approach for the cloudlet.

1. Introduction

As smartphones and tablets are gaining more and more capabilities (in terms of CPU power, network connectivity and sensors), they are becoming the preferred device for daily tasks such as browsing the web and e-mail. Although high-end smartphones are equipped with dual- or quadcore processors, their processing capabilities remain an order of magnitude lower than their desktop counterparts, due to the limited battery capacity, form factor and passive cooling. Therefore, these mobile devices still fall short to execute future resource intensive multimedia applications, that involve real-time processing of video or audio, such as object recognition or mobile augmented reality (Jäppinen et al., 2013).

To enable such resource-intensive applications, the concept of cyber foraging was introduced by Satyanarayanan (2001), where parts of the application are offloaded to nearby resources – called surrogates – connected via a wireless network to the mobile terminals. Such surrogates are recently defined as "cloudlets" (Satyanarayanan et al., 2009), virtualized infrastructure co-located with the wireless access point, which allows mobile devices to offload applications to nearby resources in the LAN network.

Today many such cyber foraging systems exist (Kovachev and Klamma, 2012), offloading applications either on a method (Cuervo et al., 2010; Kristensen and Bouvin, 2010), a software component (Giurgiu et al., 2009; Verbelen et al., 2011; Ra et al., 2011) or a virtual machine (VM) level (Satyanarayanan et al., 2009; Chun et al., 2011). Although a lot of research has been done on profiling and partitioning applications and decision algorithms to decide when to offload, two important issues still remain.

First, almost all systems focus on the scenario where one mobile device offloads to one surrogate, called the cloudlet. However, in a realistic scenario multiple devices will be connected to the wireless network, some of them requiring resources to offload to (i.e., mobile devices or tablets) and other devices offering resources (i.e., servers or laptops). In this paper we present a cloudlet management platform, that extends the cloudlet to all devices in the network. This allows taking decisions optimizing behavior for all devices in the network, taking into account shared resources such as the network bandwidth, instead of optimizing on each device separately.

In addition to cloudlets solely consisting of fixed infrastructure (e.g., a corporate cloudlet), also ad hoc cloudlets can be formed with all devices available in the LAN network (e.g., resources

* Corresponding author.
E-mail address: tim.verbelen@intec.ugent.be (T. Verbelen).

1084-8045/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.jnca.2013.12.002

discovered in the home network or within a railway carriage), as depicted in Fig. 1. The platform can also easily be extended towards the cloud, allowing application components to be distributed in a 3-tier fashion: locally on the mobile device, nearby on the cloudlet infrastructure or on a distant datacenter.

A second issue is that current cyber foraging systems aim to optimize a performance goal from a user perspective, such as energy usage, processing time or throughput. However, from a developer perspective, a more important goal is to achieve good performance on a wide range of devices, in which case "good" performance is an application specific metric. Such a metric can often be formulated as a number of constraints (e.g., maximum allowed execution time for a specific method call), especially in applications with real-time requirements.

To address this issue, our cloudlet platform manages applications on the component level, where the application developer declares application components with their offered functionality, configurable parameters and performance constraints. Application components are managed and configured at runtime, and can be distributed among resources in the cloudlet depending on the current context, in order to achieve the desired performance.

We present an overview of the architecture of our cloudlet platform, which is based on an earlier prototype we proposed in Verbelen et al. (2012). The work is further extended by including dynamic runtime adaptation to the framework, which is based on the OSGi Allaince (2009b) specification . In order to ease development of component-based mobile applications, and to allow the developer to efficiently declare application specific metrics and constraints, we also introduce a programming model based on annotations, minimizing the burden put on the application developer, and integrated our middleware in the popular Android platform. To show the need and effectiveness of the platform, we focus on an augmented reality use case.

The remainder of this paper is structured as follows. The next section gives an overview of related work in the field of application offloading and cloudlets. Section 3 presents our mobile augmented reality use case that illustrates the need for a component-based approach. This use case has been fully implemented to evaluate the cloudlet platform. In Section 4 we propose a general architecture for a cloudlet framework, and in Section 5 we discuss in more detail the implementation of a prototype based on OSGi, and the integration with the Android OS. To ease component-based application development supporting the platform, an annotation-based programming model and corresponding developer tools are presented in Section 6. In Section 7 we show the need and effectiveness of our approach for the mobile augmented reality use case, while Section 8 concludes this paper and discusses future work.

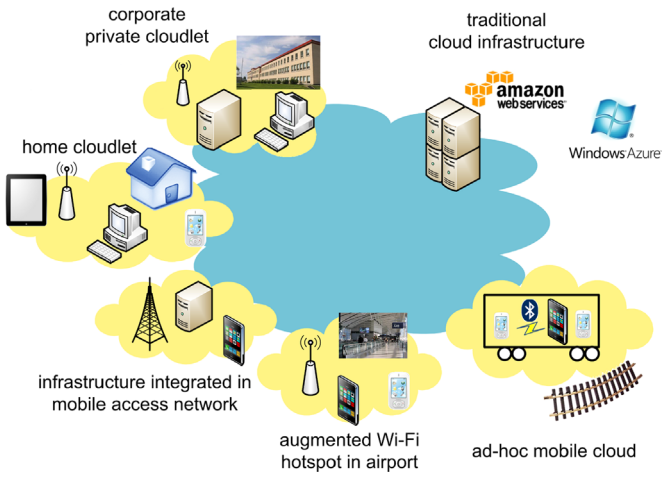

Fig. 1. Static cloudlets can be provided by a corporation in a corporate cloudlet, or by a service provider in the mobile network. Ad hoc cloudlets can be formed in the home network or within a railway carriage.

2. Related work

Offloading computation from mobile devices to remote surrogates, known as cyber foraging (Balan et al., 2002), has been subject for over a decade. Different approaches exist (Kovachev and Klamma, 2012), offloading parts of the application either using a method, component or virtual machine level granularity. Kristensen and Bouvin (2010) propose Scavenger, a cyber foraging system in Python which outsources Python methods. A scheduler predicts the execution time of a method as a function of the input parameter values and/or sizes using monitoring information from the previous executions, and selects the device offering the shortest execution time. MAUI (Cuervo et al., 2010) offloads methods of Microsoft.Net applications in order to save energy. At initialization time, MAUI measures the energy characteristics of the device, and at runtime the application and network characteristics are monitored. From this monitoring information, a call graph of the application is constructed, where the nodes are weighted with the energy used for processing the task, and the edges are weighted with the energy needed for sending the arguments and return value over the network. An integer linear programming (ILP) solver then decides at runtime whether or not a method is offloaded, by calculating a program partitioning that minimizes the smartphone's energy consumption, subject to latency constraints.

Using software components as unit of deployment rather than methods results in less fine-grained optimization options, but reduces the overhead of monitoring. Giurgiu et al. (2009) and Verbelen et al. (2011) use OSGi components offloadable units. Monitoring information is used to build a graph model of the application components, from which graph cutting algorithms calculate the optimal deployment, i.e., minimizing the required bandwidth or execution time. Odessa (Ra et al. (2011)) is a cyber foraging system that focuses on data flow applications. The system is built on Sprout, a distributed stream processing framework that provides a programming model to develop parallel processing pipelines, hiding much of the complexity of parallel and distributed programming from the application developer. Odessa uses a greedy algorithm that estimates the bottleneck in the processing pipeline, and tries to optimize the makespan and throughput the application by exploiting offloading and data-parallelism.

Goyal and Carter (2004) propose the usage of virtual machine technology as surrogates for remote execution, which provides isolation and security. Because the deployment of virtual machines in a cloud can lead to high WAN latencies, Su and Flinn (2005) present Slingshot, where the VMs are co-located with the wireless access point. Satyanarayanan et al. (2009) proposed the concept a of cloudlet: a trusted, resource-rich computer or a cluster of computers well connected to the Internet and available for use by the nearby mobile devices. Cloudlets offer their resources to mobile devices by dynamic VM synthesis, where small VM overlays are sent to the cloudlet from which a complete VM is created running the mobile application. Because of the coarse granularity of VMs, these systems do not contain any partitioning logic, but rather an all-or-nothing approach is used, executing the application either locally or completely on the VM using a thin client protocol. In order to enable some more fine grained partitioning, Chun et al. (2011) present CloneCloud, where a set of possible partition choices are made at build time, using static code analysis. Different binaries of the application are generated and dedicated VM instructions are added at migration points for selected

methods. At runtime, a clone VM is instantiated at the server side, and the application transparently switches between execution at the device or at the clone. In an offline profiling stage, monitoring information is collected for each partitioning identified, which is used to determine the best partitioning at runtime.

All these cyber foraging systems address the situation where the application is distributed between one mobile device and the cloud. However, in a wireless network context, network bandwidth is a shared medium, and thus there is a need for a global management entity that optimizes the bandwidth for all users. Also, all existing cyber foraging systems try to optimize generic metrics, such as energy usage, processing time, and it is often left undefined what should happen when no surrogate is available. From the application developer viewpoint, one would prefer to optimize the perceived QoS of the end user, and to gracefully degrade when no remote resources are available, or conversely, treat the situation when parts are offloaded as a quality enhancement on top of a more basic application functionality. To achieve this, the offloading framework should also be able to reconfigure the application at runtime.

To tackle these issues, we present a cloudlet middleware platform that handles both distribution as well as configuration of applications, and manages resources discovered in the network. As pointed out by Kovachev and Klamma (2012), a component level approach allows for more fine-grained execution adaptation, rather than managing applications on a VM-level as defined by Satyanarayanan et al. (2009). To facilitate application development on our framework, we present an intuitive programming model based on code annotations, and integrated the middleware in the popular Android platform.

In short, the main contributions of this paper are as follows: (i) a middleware platform for discovering and managing devices in the local network, bundling their resources as a cloudlet, (ii) a component-based offloading framework, able to distribute application components on the resources in the cloudlet, as well as configure them at runtime, in order to optimize the QoS perceived by the end user, and (iii) an annotation-based programming model, facilitating the development of mobile cloudlet applications and providing integration with the Android platform.

3. Use case: mobile augmented reality

As a use case, we present a mobile augmented reality application featuring markerless tracking as described by Klein and Murray (2007), combined with an object recognition algorithm presented in Lowe (2004). The application is shown in Fig. 2. On the right a greyscale video frame is shown with the tracked feature points, from which the camera position is estimated. The left shows the resulting overlay with a 3D object, and a white border around the recognized book.

Starting from the source code from Klein and Murray (2007), we have redesigned the parallel tracking and mapping algorithm algorithm as a set of loosely coupled components. These are wrapped in Java code, in order to obtain software components that are able to run on multiple platforms and hence can be offloaded by our platform, as is discussed more thoroughly in Section 5. We also extended the application with an object recognition component, as shown in Fig. 3.

VideoSource : The VideoSource fetches video frames from the camera hardware. These frames are analyzed by the Tracker, and rendered together with an augmented reality overlay by the Renderer.

Renderer : Each camera frame is rendered on screen together with an overlay of 3D objects. These 3D objects are aligned according to the camera pose as estimated by the Tracker.

Tracker : The Tracker analyses video frames and calculates the camera pose by matching a set of 2D image features to a known map of 3D feature points. The map of 3D points is generated and updated by the Mapper.

Mapper : From time to time the Tracker sends a video frame to the Mapper for map generation and refinement. By matching 2D features in a sparse set of so-called keyframes, the Mapper can estimate their 3D location in the scene and generate a 3D map of feature points.

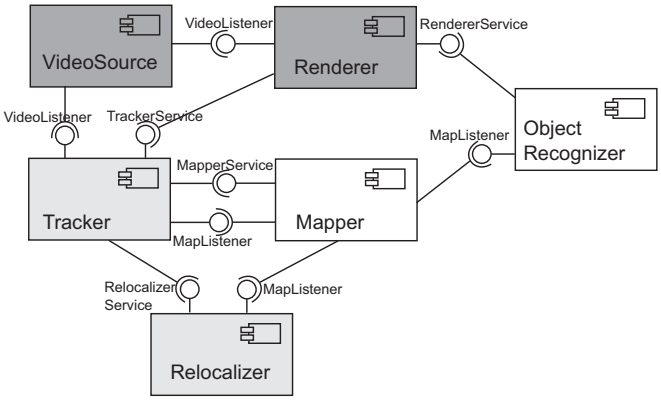

Fig. 3. The component diagram of the AR application. The VideoSource and the Renderer (dark grey) are fixed on the mobile device. The Relocalizer and Tracker (grey) have real-time constraints (≤ 50 ms).

Fig. 2. The augmented reality application tracks feature points in the video frames (right) to enable overlaying of 3D objects (left).

Relocalizer : When no 2D image features are found in the video frame, the Relocalizer tries to relocate the camera position until tracking resumes.

Object Recognizer : In the keyframes of the Mapper the Object Recognizer tries to locate known objects. When an object is found, its 3D location is notified to the Renderer that produces an overlay.

These components are not only very CPU intensive, some of them also have strict real-time constraints. The VideoSource and the Renderer have to be executed on the mobile device, as they access device specific hardware. In order to achieve an acceptable performance, the Tracker and the Relocalizer should be able to process frames within 30–50 ms, which translates into a frame rate of 20–30 frames per second. As the Mapper runs as a background task, constantly refining and expanding the 3D map, this component preferably runs on a device with considerable CPU resources. However the Mapper does not impose strict real time requirements. The Object Recognizer also has more relaxed requirements, as delays in the order of a second before recognizing an object are still tolerable to achieve an acceptable user experience.

The goal now is to run this application on a mobile device, while meeting all the required constraints, which have to be declared by the application developer. This use case shows that offloading the whole application to the cloudlet on a VM level is not sufficient, as sending frames to the cloudlet for processing and receiving rendered frames in return would require too much bandwidth, which is a scarce resource in the current wireless access networks. Therefore we propose a cloudlet architecture that will manage the application on a component level, being able to distribute application components within the cloudlet or to other cloudlets.

In addition to component (re)distribution, QoS can also be assured by dynamically reconfiguring components, as shown in Li (2012). By adopting a component-based approach for the cloudlet, real-time constraints of applications can be achieved using combined dynamic reconfiguration and distribution. In our use case for example, the developer defined a parameter configuring the number of tracked features, which influences the frame processing rate and which can be adapted at runtime.

4. Cloudlet architecture

We adopt a component-based cloudlet architecture as proposed in Verbelen et al. (2012), which is shown in Fig. 4. In this paper, we define a cloudlet as a collection of interconnected resources in each other's vicinity, for example all devices connected by a LAN network (e.g., your mobile phone and your laptop connected to your home WiFi network). Our architecture has three interrelated management levels: the component level, the node level and the cloudlet level.

Components (units of deployment specified by their providing and required interfaces Szyperski, 2002) are managed by an *Execution Environment* (*EE*) that can start and stop components, resolve component dependencies, expose provided interfaces, etc. To support distributed execution, dependencies can be resolved with other (remote) Execution Environments. In that case, proxies and stubs are generated and the components can communicate by remote procedure calls (RPCs). Components can also define performance constraints (e.g., the maximum execution time of a method), and expose configuration parameters to the EE. By monitoring the resource usage of each component, the EE can detect violations of the performance constraints and actions can be taken such as calculating a new deployment (i.e., offloading some resource intensive components) or adapting component configurations (i.e., lowering component quality).

One or more Execution Environments run on top of an operating system (OS), which in turn can run on either virtualized or real hardware. The (possibly virtualized) hardware together with the installed OS is called a node, and is managed by a *Node Agent* (*NA*). The Node Agent manages all the EEs running on the OS, and can also start or stop new Execution Environments, for example for sandboxing components. The NA also monitors the resource usage of the node as a whole, and has a view on the (maybe virtualized) hardware it runs on (e.g., the number of processing cores, processing speed, etc.).

Multiple nodes that are in the physical proximity of each other (i.e., low latency) form a cloudlet. The cloudlet is managed by a *Cloudlet Agent* (*CA*), that communicates with all underlying Node Agents. Cloudlet Agents of different cloudlets can also communicate with each other, for example to migrate components between cloudlets. Within a cloudlet, the node with the most resources is chosen to host the Cloudlet Agent. The Cloudlet Agent has a global overview of all available resources, and is able to calculate a deployment, which should be globally optimal for all devices in the cloudlet.

There are two types of cloudlets, as shown in Fig. 4: the ad hoc cloudlet and the elastic cloudlet. The ad hoc cloudlet consists of dynamically discovered nodes in the LAN network. These nodes run a Node Agent that can spawn Execution Environments to deploy components in. When nodes join or leave the cloudlet, the Cloudlet Agent will recalculate the deployments, migrating components if needed. The elastic cloudlet runs on virtualized infrastructure, where nodes run in virtual machines. Here, the Cloudlet Agent can spawn new nodes when more resources are needed, or stop nodes when too much resources are allocated. This type of

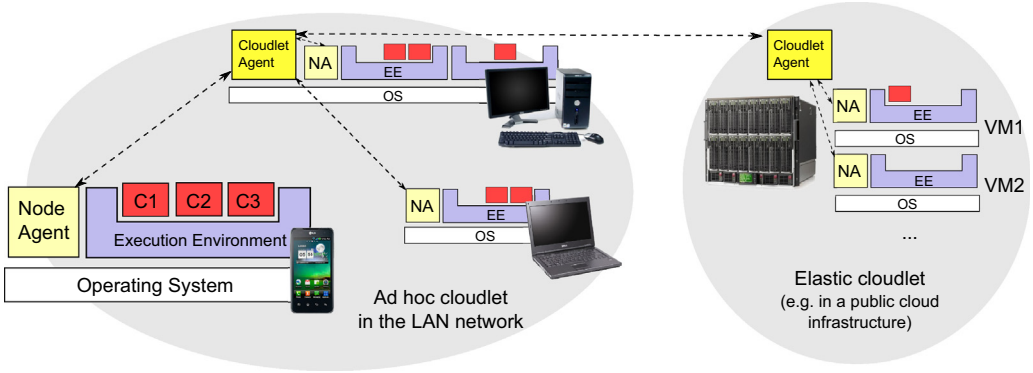

Fig. 4. The application components are distributed among nodes in two cloudlets. An ad hoc cloudlet consisting of a mobile phone, a laptop and a desktop computer, and a distant elastic cloudlet in a public cloud infrastructure.

cloudlet comes close to the VM based cloudlet envisioned by Satyanarayanan et al. (2009), the main distinction being the additional platform components running in the VM (i.e., the Node Agent and the Execution Environment) managing the applications. An elastic cloudlet could run on virtualized hardware co-located with the wireless access point, or on a private cloud within the work environment.

By also deploying the cloudlet platform in the cloud, components can also be outsourced to the public cloud, which allows applications to be outsourced in a 3-tier fashion: hardware dependent components are deployed on the mobile device, latency constrained components are distributed on a nearby cloudlet and unconstrained components, exhibiting challenging CPU and/or data storage requirements, can be deployed on EEs in the public cloud.

5. Cloudlet platform design and implementation

The prototype implementation of the cloudlet platform is realized in OSGi (2009b), an industry standard module management system in Java. This technology which was selected as the OSGi standard already specifies the software component concept, and the OSGi platform can run on multiple underlying platforms. In addition, a substantial amount of middleware building blocks (e.g., Configuration Admin and Declarative Services) are standardized in the OSGi Compendium Specification (The OSGi Alliance, 2009a). In this section, we describe in detail the design and implementation of the Execution Environment and the Node and Cloudlet Agent, together with their integration in the Android platform. First the most important OSGi concepts are highlighted, on which the cloudlet platform is based.

5.1. OSGi

The OSGi core specification defines a service oriented module management system for 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 needs to call another service, the service registry is queried for available implementations. The portability of Java enables the execution of the same code on different platforms and architectures, facilitating remote execution and code migration. OSGi also allows to incorporate native libraries in a bundle, which are loaded at runtime depending on the underlying platform. Because OSGi was first designed for embedded devices, the incurred overhead is limited.

The OSGi module management system constitutes the core of the Execution Environment of the cloudlet platform. In addition to the core specification, we also use four compendium specifications: Declarative Services, the Configuration Admin Service, the Metatype Service and Remote Service Admin.

5.2. Execution Environment

An overview of the Execution Environment is shown in Fig. 5. The EE bundle proxies the components to gather monitoring information (e.g., time spent executing a method) as well as to transparently forward method calls to remote instances through the Remote Service Admin. The latter component implements a remote procedure call protocol, such as R-OSGi (Rellermeyer et al., 2007). Components come with three descriptors: the metatype descriptor exposes the configurable parameters, the service description declares the offered and required service interfaces and the SLA descriptor defines the imposed constraints. The Declarative Services bundle ensures that component interfaces are registered with the OSGi runtime, and the component is bound to all its dependencies. The Metatype and Configuration Admin service allow the EE to discover the available configuration options and to set these configurations dynamically to meet the imposed constraints.

The Declarative Services specification presents a declarative model for publishing, finding and binding OSGi services. Instead of registering and looking up services programmatically in the source code, this is now done at runtime by the OSGi framework using an XML description that is bundled with the component, which describes the provided service interfaces, the service dependencies and how these dependencies should be bound to the component. For lifecycle management the developer can also define methods to be called when a component is activated or deactivated. This provides a simplified programming model to the developer, which has no more code dependencies to the OSGi core APIs, and does not have to handle with the complex dynamic service concept. An example of such a component description XML is shown in Listing 1.

Listing 1. An example of a snippet of a component description, defining the Tracker component. The Tracker component implements the TrackerService and VideoListener interfaces, and needs a reference to the MapperService, which is injected by the framework using the bind/unbind methods.

```
<component name="ptam.tracker.Tracker">
   <implementation class="ptam.tracker.Tracker"/>
   <service>
      <provide interface="ptam.tracker.api.TrackerService"/>
      <provide interface="ptam.video.api.VideoListener"/>
  ...
   </service>
   <reference name="mapper"
         interface="ptam.mapper.api.MapperService"
  ...
         bind="setMapper"
         unbind="unsetMapper"/>
    ...
</component>
```

The ConfigurationAdmin (CA) decouples a component from its configuration parameters. The CA maintains a persistent repository of configuration data (a dictionary of key-value pairs) for all components. When a component is started, the CA provides the necessary parameters to correctly configure the component. When the configuration data is changed, components are updated accordingly, allowing at runtime reconfiguration.

Because the EE has to be able to adapt the component configuration at runtime using the ConfigurationAdmin, it should also know which configuration parameters are available and which values are allowed. This is where the Metatype Service comes in, which allows developers to describe attribute types in an XML format. An example of a metatype XML is shown in Listing 2. The metadata consists of an Object Class Definition (OCD), which defines a number of Attribute Definitions (ADs), and a Designate, which binds the OCD to a component.

Listing 2. An example of a metatype information for the Tracker component that marks the number of tracked points as a configurable parameter.

```
<MetaData>
      <OCD description="PTAM Tracker Configuration"
     name="PTAM Tracker"
     id="ptam.tracker.TrackerOCD">
           <AD name="Fine Points"
              id="tracker.finePoints"
              default="1000"
      min="200"
      max="1000">
           </AD>
      </OCD>
      <Designate pid="ptam.tracker.Tracker">
           <Object ocdref="ptam.tracker.TrackerOCD" />
      </Designate>
</MetaData>
```

To be able to distribute components among multiple resources in the network, we use R-OSGi as a provider for our Remote Service Admin, which enables remote service binding and remote service calls.

In order to monitor all application components, the Execution Environment bundle creates a proxy for each service interface provided by a component. Using the OSGi hooks API from the core specification, the original service is hidden for other components, that can only bind to the proxied one. This way, all service calls pass through the proxy, which is able to monitor all service calls. The proxy forwards the call to an actual component interface, either locally or remote via R-OSGi. Using the Metatype service, the EE can discover the configurable parameters for each component, and modify these using the Configuration Admin. The configuration is adapted in order to meet constraints that are declared by the developer in the constraints descriptor as shown in Listing 3.

Listing 3. An example of a constraint description for the Tracker component. A video frame should be processed within 50 ms.

```
<sla>
  <service name="ptam.video.api.VideoListener">
    <method name="processFrame">
      <constraint type="maxTime">50</constraint>
    </method>
  </service>
</sla>
```

Here a constraint is defined that a video frame should be processed by the Tracker component within 50 ms. The Execution Environment implements a feedback loop that will constantly monitor the time to process a video frame. When the processing time exceeds 50 ms, the Tracker configuration parameters are changed in order to lower the processing time. The EE also notifies the Node Agent, which will ask the Cloudlet Agent whether the components also need to be offloaded to enhance the application quality.

In order to create an application that exploits the cloudlet platform, the developer needs to design the application as a number of OSGi bundles. For each bundle he has to define the component descriptor that describes the provided and required services of the component. Optionally, the developer can also declare the configurable parameters in a metatype descriptor, and required constraints in a constraint descriptor. Because this can be a tedious and error prone task, we propose an annotation based programming model in Section 6 to alleviate development.

5.3. Node and cloudlet agent

The Node and Cloudlet agents are also implemented as OSGi bundles, running on an OSGi runtime in a separate process, as shown in Fig. 6. When an EE is started, it registers with the Node Agent running on that node via R-OSGi on localhost.

When first started on a device, both the Node Agent and the Cloudlet Agent bundles are active. Other devices are discovered using jSLP, a Java implementation of the SLP discovery protocol (Guttman et al., 1999). When another CA is discovered in the LAN network, the CA on one of the two devices (the weakest of the two) is stopped, and the remaining CA becomes the master of the cloudlet. If the connection to the CA is lost, the Cloudlet Agent bundle is started again locally. This way, a cloudlet can be formed in an ad hoc manner, without the need for virtualized hardware. When the discovered Cloudlet Agent is not deployed nearby, both CAs communicate as peers, allowing for inter-cloudlet component offloading.

5.4. OSGi and Android integration

To deploy our cloudlet platform on the Android platform, the platform needs to strictly comply with Androids application model. Android applications are written in Java and compiled to Dalvik bytecode (which run on the Dalvik Virtual Machine), and 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 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) without blocking the user interface. Content providers are used for managing a shared set of application data, and broadcast receivers are small components that respond to system-wide broadcast announcements (e.g., announcement when running low on power).

The OSGi runtime with the Node and Cloud Agent are embedded in a separate Cloudlet Android application that starts an Android service at boot time, which allows the Node Agent to run in the background. The Execution Environment is embedded within an Android application as depicted in Fig. 7. Within the Android application, components are identified that should be offloadable or that should be monitored and configured by the EE. The Android application calls these components by looking up their services with the OSGi runtime, and can also register services to enable callbacks from the components. Because the offloadable components can be executed both on the Android device as on a remote server, the code is compiled in both the Dalvik bytecode format and the regular Java bytecode format.

Figure 7 also shows how the use case components are embedded in the Android application. The Renderer and VideoSource components reside within the Android activity, as they need to render to screen, or need to access to the Android internals to fetch video frames from the hardware. They expose their interfaces to the other components through OSGi and the EE. The other components such as the Tracker and the Mapper are packaged within the .apk as separate jars, which allows these components to be offloaded at runtime, and are deployed on the OSGi runtime.

The resulting application acts as a regular Android application, and no additional SDK is needed. When the application starts, the EE is created which registers with the Node Agent running on the device in the background. However, a huge burden is put on the application developer when he wants to develop such a component-based application. Necessary steps to take include: identifying the components, implementing these separately as OSGi-bundles together with the accompanying descriptors, embedding an OSGI runtime together with the EE, and building and packaging the components as an Android application. To simplify this process, a lightweight programming model is presented in the following section.

6. Programming model

To ease the development of cloudlet enabled Android applications, we present a straightforward programming model based on Java annotations. These annotations are analyzed in a preprocessing step in the build process, from which the required XML descriptions are generated. In the case of an Android application, the build system will also generate the code necessary to embed the OSGi runtime and the cloudlet platform, and create the desired components.

Listing 4 shows the annotated source code of the Tracker class, from which the Tracker component is generated, as well as the descriptors as described in Section 5. Classes annotated by `@Component` are handled as component implementation classes, providing all implemented interfaces. A reference to another service is created by declaring a variable of the desired type and annotate it with the `@Reference` annotation. This will generate a set and unset method to inject a reference to the required service at runtime, and create the reference attribute in the XML description. Configurable properties are annotated by `@Property`, from which the metatype description is generated. To declare constraints on service methods, they can be annotated with the `@Constraint` annotation, describing the type constraint and the imposed threshold.

Listing 4. An example of annotated Java source file from which a component is generated, together with the required XML descriptors.

```
package ptam.tracker;

@Component
public class Tracker implements TrackerService, VideoListener {

  @Property(min=200, max=1000)
  private int finePoints = 1000;

  @Reference(policy=dynamic)
  private MapperService mapper;

  @Constraint(maxTime=50)
  public void processFrame(byte[] data) {
    // process frame
    ...
  }
  ...
}
```

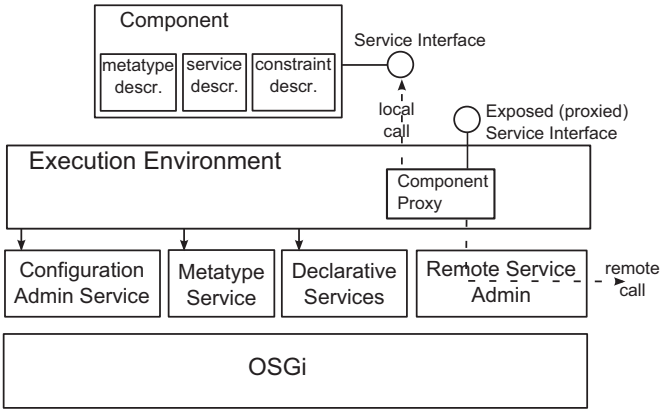

Fig. 5. The Execution Environment proxies all component interfaces in order to monitor all (local and remote via RemoteServiceAdmin) service method calls. The Configuration Admin and Metatype service enable the EE to dynamically configure components, which are defined using the Declarative Services specification.

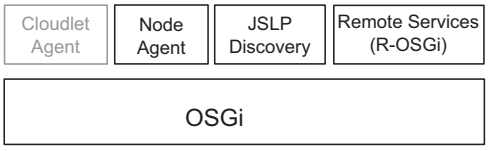

Fig. 6. The Node Agent and Cloudlet Agent bundles run on an OSGi runtime in a separate process. The Cloudlet Agent bundle is only present when the current node is also the master of the cloudlet.

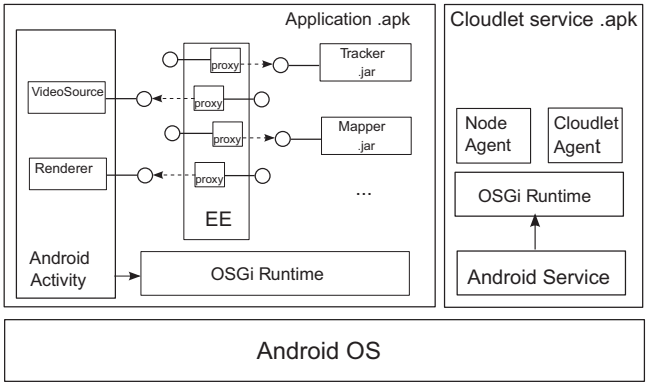

Fig. 7. On the Android platform a separate application hosts the Node Agent and the Cloudlet Agent bundles, that run in a background process using an Android Service. Applications having offloadable and/or configurable components deploy these on an OSGi runtime hosting the EE, which proxies the provided interfaces. Components requiring the Android internals (e.g., drawing to screen) are embedded in the Android activity and also provide their interfaces through the EE.

The resulting build process is shown in Fig. 8. First the source files are preprocessed and XML descriptors are generated from the detected annotations. In the second step, the source code of the components is compiled and packaged together with the XML descriptors. Finally the build system combines the application components with the middleware components and packages the resulting Android .apk.

7. Evaluation

To show the need for a cloudlet platform presented in this paper, we implemented the augmented reality use case presented in Section 3 using the proposed component model. The AR code was executed on an ad hoc cloudlet consisting of a mobile device and a laptop connected via WiFi. The laptop is equipped with an Intel Core 2 Duo CPU clocked at 2.26 GHz. As mobile device we use a HTC Desire, with a single core Qualcomm 1 GHz Scorpion CPU, and an LG Optimus 2 × powered by a dual core Nvidia Tegra 2 CPU, also clocked at 1 GHz. Both devices capture camera frames at a 800 × 480 resolution, which is the same size as their screen resolution. The results show the benefits of offloading resource intensive components such as the Mapper and Object Recognizer, and the need for dynamically configurable components in the case of the Tracker.

7.1. Offloading resource-intensive components

In order to address the limited CPU resources of mobile devices, resource-intensive components can be offloaded from the mobile devices to the laptop in the network. However, when a component is offloaded, this also affects the bandwidth usage, as this involves remote method calls. In our use case, one can intuitively identify the Mapper and Object Recognizer as components suitable for offloading, as these have high CPU requirements, but the required bandwidth remains low as not every frame needs to be processed, and there are no strict time constraints. The Tracker component on the other hand has to process each camera frame at a sufficient frame rate, which makes this component less suitable to offload, as this would result in a high bandwidth.

To illustrate the effect of component distribution on the bandwidth utilization, we conducted the following experiment. We connected the LG Optimus 2 × device to the laptop, both running the cloudlet framework with the Cloudlet Agent hosted on the laptop, thereby forming an ad hoc cloudlet using the discovery process described in Section 5.3. For this experiment we used a fixed network connection over USB between the devices, in order to be able to illustrate the high bandwidth requirements of offloading the Tracker component. On the mobile device we start the Android application with an embedded Execution Environment hosting the AR application components, and at the server side also an EE is started to offload components. During execution, the Node Agents

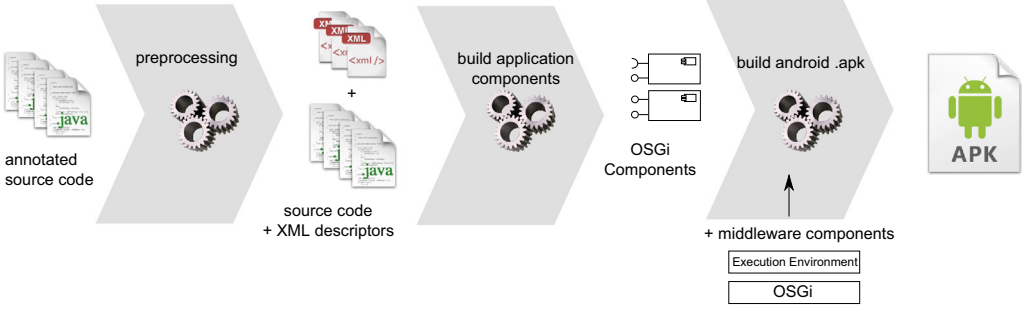

Fig. 8. Overview of the steps done by the build system. Annotated source code is preprocessed and application components are generated. These are bundled with the middleware components to create the resulting Android .apk which can be deployed on the Android OS.

monitor the CPU usage on the devices, as well as the network traffic. The results are shown in Fig. 9.

At first, all processing is done at the mobile device, resulting in a close to 100% CPU load. After 40 s, we instruct the Cloudlet Agent to migrate the Object Recognizer and Mapper components to the laptop. This results in bandwidth usage two peaks as well as in a server side CPU load peak, as the components have to be initialized there, and the state has to be synchronized. Also, first all local method calls that are in progress have to finish before the component is actually offloaded. Once the two components are offloaded, the CPU usage at the mobile device is less saturated, a peak in bandwidth and server side CPU usage occurs each time a frame is added to the map. After 120 s we instruct the Cloudlet Agent to also offload the Relocalizer and Tracker components. Since now all processing is done at the server side, we see a constant increase in CPU usage at the server, while the CPU usage of the mobile device drops. However, as now all (raw) image frames are sent over the network, around 40 Mbps is required to achieve a frame rate of 15 frames per second. For every 5 s also a small peak in bandwidth usage can be seen. This is the monitoring information that is collected over the network by the Cloudlet Agent, which consumes about one kilobyte.

This experiment illustrates the need for offloading at component granularity, allowing to trade-off between CPU usage and bandwidth usage. In our use case the Tracker component is clearly less suitable for offloading, as such a high bandwidth is not readily available in current wireless networks.

To automatically decide at runtime which components should be outsourced and which should be executed on the mobile device, algorithms can be used based on the monitoring information gathered by the Execution Environment. This decision problem can, for example, be modelled as a graph partitioning problem where the application is represented as a weighted component graph with, for example, CPU usage or energy usage as component weights, and communication between components as edge weights. Deciding which components to offload is then calculated using a graph cutting algorithm, for example a min-cut based greedy algorithm as proposed by Niu et al. (2013), or heuristics using Simulated Annealing or Kernighan–Lin as presented in Verbelen et al. (2013).

7.2. The benefits of offloading

When offloading components to resource-rich infrastructure, another and more important benefit (since it directly relates to the application quality as perceived by the end-user), besides a decrease in CPU usage on the mobile device, is a speedup in execution time. As the Execution Environment is able to monitor the execution time of the different methods provided by a component as discussed in Section 5.2, we can compare the execution time of local and remote execution of the Mapper and Object Recognizer components.

Each time when a new keyframe is added to the map, the Mapper optimizes all positions of the points in the map. This optimization problem scales with the total number of keyframes in the map. Figure 10 shows the time needed for this map refinement as a function of the keyframes in the map for one run, both executed locally on the LG Optimus 2×, and remote when the Mapper component is outsourced to the laptop over a WiFi network. As the number of keyframes increases, the refinement process takes up to 10 s on the mobile device, while remote execution takes less then a second. Overall we witness a speed up of a factor 10 when outsourcing the laptop.

For the Object Recognizer, larger differences are perceived. When performing an object search on a videoframe, this takes up to 30 s on the HTC Desire, and around 18 s on the LG Optimus 2×. When outsourcing the Object Recognizer to the laptop, the processing time is reduced to an average 1.5 s.

Next to a decrease in execution time, offloading components may also result in a decrease in memory usage, hence relaxing memory requirements for the mobile device. In the use case studied here, we indeed observe a severe reduction in memory activity on the mobile device when the Mapper and Object

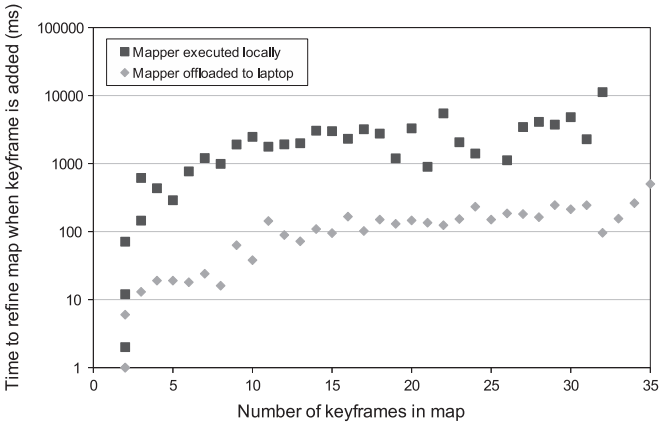

Fig. 10. The time to refine the map when a new keyframe is added, as a function of the total number of keyframes in the map. Offloading the Mapper component results in a speedup factor of 10.

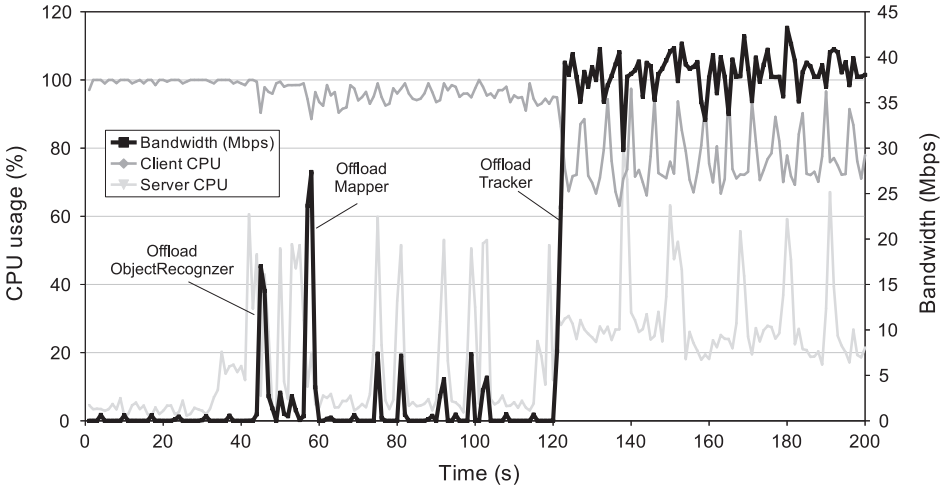

Fig. 9. Offloading components from the mobile device to the laptop lower the CPU usage at the mobile device, but increase bandwidth usage.

recognizer are offloaded to the laptop. However, as memory is managed by the JVM internally through garbage collection, it is difficult to account for memory usage per component at runtime, hence making memory usage a less suitable metric for runtime decision taking.

7.3. The need for dynamic configuration

The performance of the Tracker component, which has to process all video frames, is critical to the user experience, as too few frames processed per second deteriorates the application quality. However, as shown in Fig. 9 in this case offloading is not an option, as not enough bandwidth is available to send and receive the video frames in a timely manner. The Tracker exposes one configuration parameter representing the maximum number of feature points that can be tracked. A high number of points will increase the tracking accuracy, making the Tracker more resilient to fast camera movements or motion blur, at the cost of more processing power.

For both mobile devices, the average processing time (and standard deviation) to track one video frame as a function of the number of tracked features is shown in Fig. 11. It is clear that in order to meet the constraint of 50 ms to process a frame, the number of tracked features for the HTC desire has to be limited to 200, while the LG Optimus 2× can process up to 450 feature points, resulting in a more robust tracking.

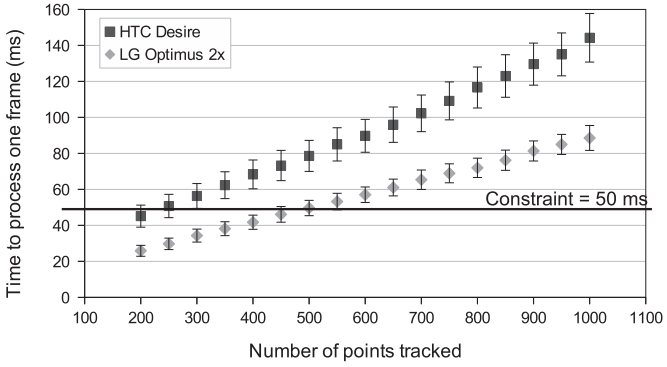

Fig. 11. In order to meet the constraint of 50 ms to process a frame, the HTC Desire can process at most 200 feature points per frame, while the LG Optimus can handle 450 feature points.

In order to illustrate the dynamic adaptation of the configuration, we implemented a negative feedback loop in the EE bundle: when the moving average of the monitored execution time of tracking one frame is smaller than 50-t ms, the number of tracked feature points is increased, where $t=10$ is chosen. Similarly when the execution time rises above 50 ms, the amount of tracked feature points is decreased. This enables the framework to automatically adjust the configuration to the device, by monitoring the execution time at runtime.

The effect of this feedback mechanism is shown in Fig. 12. Here we execute all AR components locally on the LG Optimus 2× device, and the tracking parameter is configured dynamically. The processing time per frame is shown as well as the value of the configurable parameter (=maximum number of feature points tracked).

The initialization of the map (1) results in high processing times, and thus the value of the parameter is reduced to the minimum. Once the tracking is started, the parameter value is increased to have the best tracking accuracy possible within the 50 ms imposed. Indeed, as expected from Fig. 11, the LG Optimus 2× is able to track 450 feature points.

When the tracking is lost due to a too quick camera movement, no feature points are found, and the applications starts relocalizing (2). As no feature points are found, the processing time which lowers the parameter value is increased. When tracking resumes (3), the processing time is too high due to the increased parameter, and the parameter value is adapted again to meet the imposed constraint.

This shows that the framework not only adapts the configuration depending on the device capabilities at initialization, but can also adapt to different application scenarios at runtime. In this case it is beneficial to increase the number of tracked feature points when tracking is lost, as this gives more robust tracking when the relocalizer finds a reasonable estimate of the camera position. Once tracking is stable again, the number of feature points can be reduced to meet the imposed framework.

These results illustrate the benefits of dynamic configuration adaptation, but the implemented solution is very application specific. Future work is needed to search for more generic solutions that adapt the application without a priori knowledge, for example using self-learning techniques.

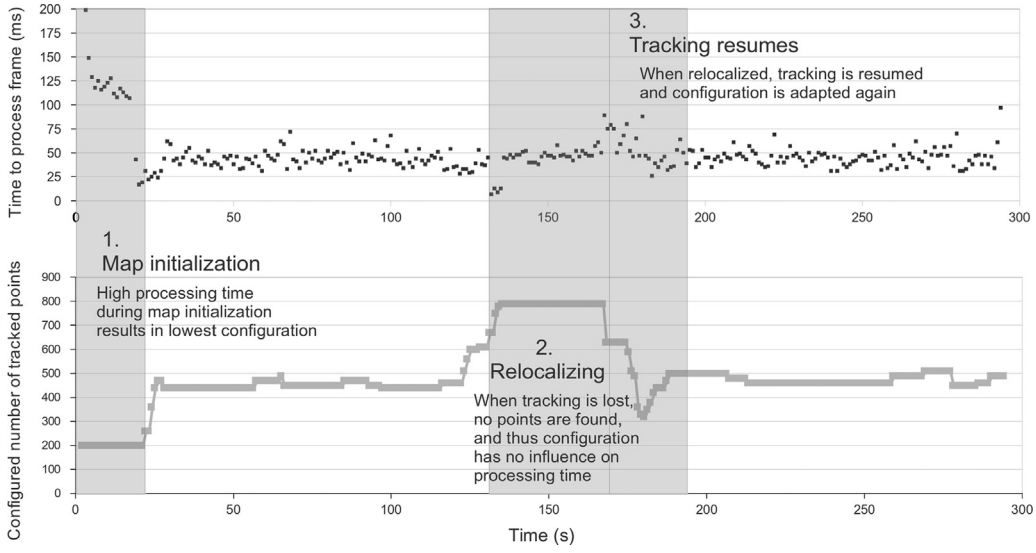

Fig. 12. The time needed to track a frame as a function of the number of feature points. Using a negative feedback loop, the framework adapts the parameter configuration at runtime in order to meet the maximum processing time constraint.

8. Conclusions and future work

In this paper, we present a cloudlet architecture that not only provides fixed infrastructure co-located with the WiFi access point, but also enables ad hoc discovery of devices in the vicinity to share resources among each other. Instead of providing infrastructure on a virtual machine level, a more fine-grained approach is presented, where the cloudlet framework manages applications on a component level. By adaptively configuring and outsourcing application components, the platform optimizes the application depending on the mobile device capabilities and the available resources in the cloudlet.

A prototype implementation of the framework based on the OSGi industry standard is presented, together with an annotation-based programming model which enables easy application development. To evaluate the cloudlet platform we implemented a component-based augmented reality application. Using a negative feedback loop mechanism, the framework dynamically adapts the configuration of the application, in order to perceive a predefined frame rate. By offloading CPU intensive components a speed up factor of 10 is achieved.

With respect to commercial deployment of such a platform, different scenarios are possible. A first scenario is to rely on the network operators to provide computing infrastructure in the edge network, for example co-located with wireless access points. This of course requires a high initial investment and management cost, and thus some kind of pay-per-use model could be adopted similarly to cloud computing, offering on-demand nearby computing resources. When the cost of providing infrastructure is too high, another option could be a FON-based scenario, where instead of users sharing their wireless connection, users also partially share their home computing resources to mobile users in the vicinity. In return, they can also use shared resources of other FON users when on the go. In order to avoid infrastructure and/or management costs, a final scenario could be a peer-to-peer scenario, where devices connect in an ad hoc manner. This scenario could be successful when all users involved gain by sharing resources, for example in a collaborative augmented reality application, i.e., where multiple devices share 3D map information to improve tracking quality. However, in all three scenarios effort has to be spent on security issues, e.g., authentication, authorization and proper isolation to prevent malicious code from executing.

As future work, more advanced algorithms are to be investigated in order to manage all resources and application components in the cloudlet. For example self-learning techniques could be used in the Execution Environment to identify which parameters to adapt, and the Cloudlet Agent could analyze monitoring information from all nodes to calculate an optimal deployment for all application components. To manage the increasing complexity, our architecture allows the hierarchical autonomic control loops, which is already applied in the topic of network management (Famaey et al., 2010).

Acknowledgement

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

This project was partly funded by the UGent BOF-GOA project “Autonomic Networked Multimedia Systems”.

References

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

Chun BG, Ihm S, Maniatis P, Naik M, Patti A. Clonecloud: elastic execution between mobile device and cloud. In: Proceedings of the sixth conference on computer systems, EuroSys ′11; 2011. p. 301–14.

Cuervo E, Balasubramanian A, Cho Dk, Wolman A, Saroiu S, Chandra R. Maui: making smartphones last longer with code offload. In: Proceedings of the 8th international conference on mobile systems, applications, and services, MobiSys ′10, ACM; 2010. p. 49–62.

Famaey J, Latre S, Strassner J, De Turck F. A hierarchical approach to autonomic network management. In: 2010 IEEE/IFIP on network operations and management symposium workshops, NOMS Wksps; 2010. p. 225–32.

Giurgiu I, Riva O, Juric D, Krivulev I, Alonso G. 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; 2009. p. 5:1–20.

Goyal S, Carter J. A lightweight secure cyber foraging infrastructure for resource-constrained devices. In: Proceedings of the sixth IEEE workshop on mobile computing systems and applications, WMCSA ′04; 2004. p. 186–95.

Guttman E, Perkins C, Veizades J, Day M. Service location protocol, version 2; 1999.

Jäppinen P, Guarneri R, Correia LM. An applications perspective into the future internet. J Netw Comput Appl 2013;36(1):249–54, http://dx.doi.org/10.1016/j.jnca.2012.08.009.

Klein G, Murray D. Parallel tracking and mapping for small ar workspaces. In: Proceedings of the 2007 sixth IEEE and ACM international symposium on mixed and augmented reality, ISMAR ′07, IEEE Computer Society; 2007. p. 1–10.

Kovachev D, Klamma R. Beyond the client-server architectures: a survey of mobile cloud techniques. In: 2012 First IEEE international conference on communications in China workshops (ICCC); 2012. p. 20–5.

Kristensen MD, Bouvin NO. Scheduling and development support in the scavenger cyber foraging system. Pervasive Mob Comput 2010;6(6):677–92 (Special Issue PerCom 2010).

Li W. QOS assurance for dynamic reconfiguration of component-based software systems. IEEE Trans Softw Eng 2012;38(3):658–76.

Lowe DG. Distinctive image features from scale-invariant keypoints. Int J Comput Vis 2004;60(2):91–110.

Niu J, Song W, Atiquzzaman M. Bandwidth-adaptive partitioning for distributed execution optimization of mobile applications. J Netw Comput Appl. 2014;37:334–47, ISSN 1084-8045, http://dx.doi.org/10.1016/j.jnca.2013.03.007.

Ra MR, Sheth A, Mummert L, Pillai P, Wetherall D, Govindan R. Odessa: enabling interactive perception applications on mobile devices. In: Proceedings of the ninth international conference on mobile systems, applications, and services, MobiSys ′11; 2011. p. 43–56.

Rellermeyer JS, Alonso G, Roscoe T. R-OSGi: distributed applications through software modularization. In: Proceedings of the international conference on middleware, Middleware ′07; 2007. p. 1–20.

Satyanarayanan M. Pervasive computing: vision and challenges. IEEE Pers Commun 2001;8:10–7.

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

Su YY, Flinn J. Slingshot: deploying stateful services in wireless hotspots. In: Proceedings of the third international conference on mobile systems, applications, and services, MobiSys ′05; 2005. p. 79–92.

Szyperski C. Component software: beyond object-oriented programming. 2nd ed.2002.

The OSGi Alliance, OSGi service platform, service compendium. Release 4, version 4.2, aQute; 2009a.

The OSGi Alliance, OSGi service platform, core specification. Release 4, version 4.2, aQute; 2009b.

Verbelen T, Stevens T, Simoens P, De Turck F, Dhoedt B. Dynamic deployment and quality adaptation for mobile augmented reality applications. J Syst Softw 2011;84(11):1871–82.

Verbelen T, Simoens P, De Turck F, Dhoedt B. Cloudlets: bringing the cloud to the mobile user. In: Proceedings of the third ACM workshop on mobile cloud computing and services, MCS ′12; 2012. p. 29–36.

Verbelen T, Stevens T, De Turck F, Dhoedt B. Graph partitioning algorithms for optimizing software deployment in mobile cloud computing. Future Gener Comput Syst 2013;29(2):451–9.