Cloudlets: Bringing the cloud to the mobile user

Tim Verbelen*, Pieter Simoens*†, Filip De Turck*, Bart Dhoedt*

*Ghent University - IBBT, Department of Information Technology
†Ghent University College, Department INWE

tim.verbelen@ugent.be, pieter.simoens@ugent.be, filip.deturck@ugent.be, bart.dhoedt@ugent.be

ABSTRACT

Although mobile devices are gaining more and more capabilities (i.e. CPU power, memory, connectivity, ...), they still fall short to execute complex rich media and data analysis applications. Offloading to the cloud is not always a solution, because of the high WAN latencies, especially for applications with real-time constraints such as augmented reality. Therefore the cloud has to be moved closer to the mobile user in the form of cloudlets. Instead of moving a complete virtual machine from the cloud to the cloudlet, we propose a more fine grained cloudlet concept that manages applications on a component level. Cloudlets do not have to be fixed infrastructure close to the wireless access point, but can be formed in a dynamic way with any device in the LAN network with available resources. We present a cloudlet architecture together with a prototype implementation, showing the advantages and capabilities for a mobile real-time augmented reality application.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Cloud Computing*

1. INTRODUCTION

Nowadays, smartphones are becoming increasingly popular. In the fourth quarter of 2011, one witnessed a 47.3 per cent increase in smartphone sales from the fourth quarter of 2010 [6]. As the capabilities of mobile devices advance (in terms of CPU power, network connectivity and sensors), people increasingly use them for other tasks such as emailing, GPS routing, Internet banking, gaming etc. Although many advances in technology, mobile devices will always be resource poor, as restrictions on weight, size, battery life, and heat dissipation impose limitations on computational resources and make mobile devices more resource constrained than their non-mobile counterparts [14]. Therefore, mobile devices still fall short to execute many rich media and data analysis applications that require heavy computation, and often also have (near) real-time constraints such as augmented reality (AR).

One solution to overcome these resource limitations is mobile cloud computing [7]. By leveraging infrastructure such as Amazon's EC2 cloud or Rackspace, computationally expensive tasks can be offloaded to the cloud. However, these clouds are typically far from the mobile user, and the high WAN latency makes this approach insufficient for real-time applications. To cope with this high latency, Satyanarayanan [13] introduced the concept of cloudlets: trusted, resource rich computers in the near vicinity of the mobile user (e.g. near or colocated with the wireless access point). Mobile users can then rapidly instantiate custom virtual machines (VMs) on the cloudlet running the required software in a thin client fashion [14].

Although cloudlets may solve the issue of latency, there are still two important drawbacks of the VM based cloudlet approach. First, one remains dependent on service providers to actually deploy such cloudlet infrastructure in LAN networks. To alleviate this constraint, we propose a more dynamic cloudlet concept, where all devices in the LAN network can cooperate in the cloudlet, as depicted in Figure 1. Next to the cloudlet infrastructure provided by service providers in the mobile network, or by a corporation as a corporate

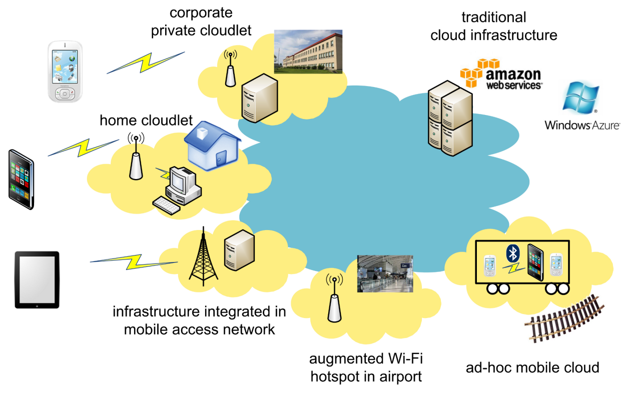

Figure 1: Static cloudlets can be provided by a corporation in a corporate cloudlet, or by a service provider in the mobile network. Ad hoc cloudlets can be formed in the home network or within a railway carriage.

cloudlet, all devices in the home network can share their resources and form a home network cloudlet. On the train, different users can also share resources in an ad hoc cloudlet.

A second drawback of VM based cloudlets is the coarse granularity of VMs as unit of distribution. Instead of executing the whole application remotely in the VM and using a thin client protocol, better performance can be achieved by dynamically partitioning the application in components [4]. Also, as resources in the cloudlet will still be limited, chances are that even the cloudlet runs out of resources when many users execute their VM simultaneously on the cloudlet infrastructure. With component offloading, a more flexible allocation of the cloudlet resources is possible, so that priority is given for latency-critical parts of the application, while non real-time parts can be offloaded to a more distant cloud.

In this paper we present a new cloudlet architecture, where applications are managed on component level. These application components can be distributed among the cloudlets. The cloudlet infrastructure is not fixed, and devices can join and leave the cloudlet at runtime. To show the need for such a cloudlet architecture, we present an evaluation using an augmented reality use case.

The remainder of this paper is structured as follows. In the next section, we will describe the augmented reality use case, its components and requirements. Section 3 describes in detail our cloudlet architecture. In Section 4 the current implementation is discussed followed by the evaluation in Section 5. Section 6 presents related work in the domain of code offloading. Finally we conclude this paper in Section 7 and discuss future work.

2. USE CASE: AUGMENTED REALITY

As a use case, we present an augmented reality application featuring markerless tracking as described by Klein et al. [9], combined with an object recognition algorithm presented in [11]. The application is shown in Figure 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.

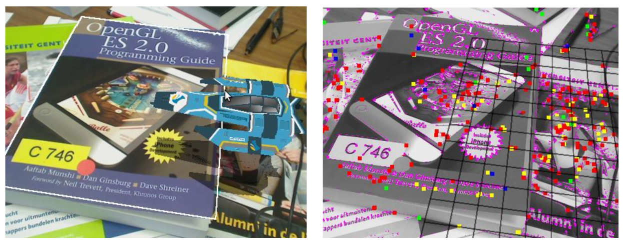

Figure 2: The augmented reality application will track feature points in the video frames (right) to enable the overlay of 3D objects (left).

We have split up the augmented reality algorithms of the application and redesigned them into the following components, as shown in Figure 3:

VideoSource The VideoSource fetches video frames from the camera hardware. These frames are analyzed by the Tracker, and rendered 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.

Relocalizer When no feature points 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 renders an overlay.

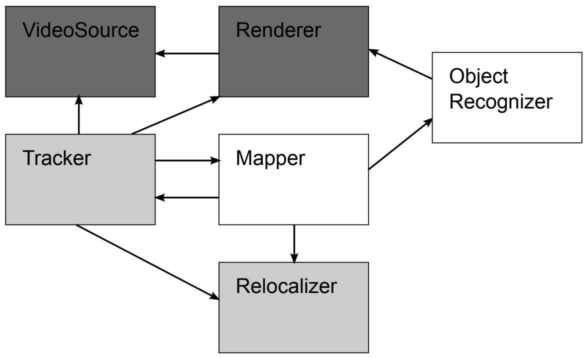

Figure 3: The different components of the AR application. The VideoSource and the Renderer (dark grey) are fixed on the mobile device. The Relocalizer and Tracker (grey) have real-time constraints ($<$ 50 ms). The Mapper and the ObjectRecognizer (white) do not have strict requirements.

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 equivalents 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 is a component that runs preferably on a device with much CPU power, but has no strict latency requirements. The Object Recognizer also has more relaxed requirements, as delays in the order of a second

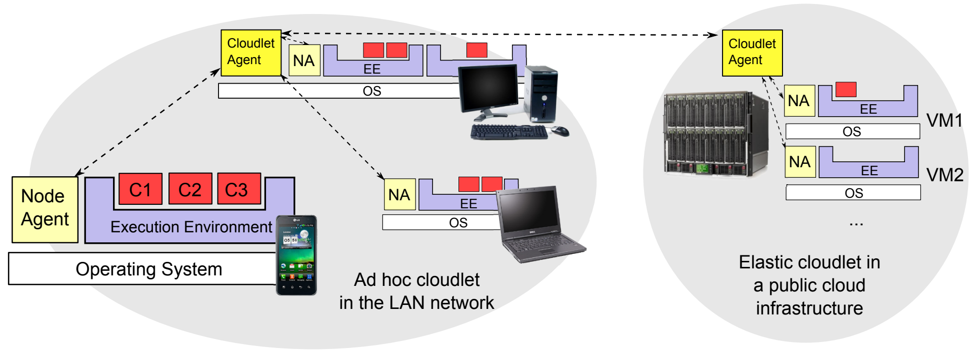

Figure 4: The application components are distributed among nodes in two cloudlets. An ad hoc cloudlet consisting of a mobile phone, a laptop and a desktop computer, and a distant elastic cloudlet in a public cloud infrastructure. All components are managed and monitored by an Execution Environment (EE). Different EEs on a node are managed by a Node Agent (NA), that on their turn communicate with the Cloudlet Agent (CA).

before relocalizing or annotating are still sufficient to achieve acceptable user experience.

The goal now is to run this application on a mobile device, while meeting all the imposed constraints. Therefore we define a cloudlet architecture that will manage the application on a component based level, being able to configure and/or distribute application components within the cloudlet or to other cloudlets.

3. CLOUDLET ARCHITECTURE

We envision the cloudlet as shown in Figure 4, with three layers: the component level, the node level and the cloudlet level.

A component is the unit of deployment that is specified by its providing and required interfaces [16]. Components 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 multiple Execution Environments run on top of an operating system (OS), which in its turn can run on both 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 info about 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.

When an EE detects a performance constraint violation, it notifies the Node Agent of the node it runs on. This Node Agent will in its turn notify the Cloudlet Agent of the cloudlet it is part of, which will then calculate a new deployment or configuration of the components. This hierarchical approach has a number of advantages. First, it is more scalable than to let all the Execution Environments decide as peers. Second, the possibly complex decision logic is not executed on the weakest devices (that signal performance constraint violations), but on the strongest device in the cloudlet. Third, the Cloudlet Agent has a global overview of all available resources, and thus is able to calculate a global optimum for all the devices in the cloudlet, rather than calculating (possibly conflicting) optima for each device separately.

There are two types of cloudlets, as shown in Figure 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 a 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 cloudlet comes close to the VM based cloudlet envisioned by Satyanarayanan [14], but with extra middleware in the VM (NA and EE) that manages the application. Elas-

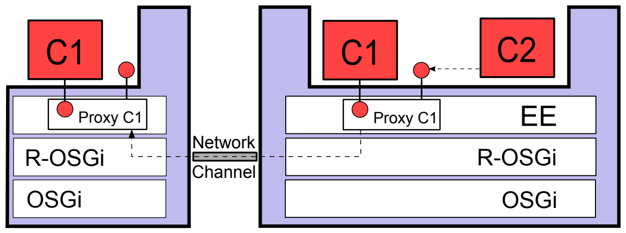

Figure 5: The internals of the Execution Environment implementation, running on top of OSGi and R-OSGi. Component C1 provides an interface that is proxied by the EE. All components requiring this interface get a reference to the EE proxy, which allows the EE to transparently monitor and migrate components.

tic cloudlets can also be used as abstraction for the public cloud, when standardized interfaces such as OCCI are used.

4. CURRENT IMPLEMENTATION

We implemented a prototype of our framework in Java, which allows it to run on most hardware such as laptops, desktops and servers, but also on Android, the most popular mobile operating system [2]. As base for the Execution Environment we adopted OSGi [17], a service oriented module management system allowing to dynamically load and unload software modules. OSGi is selected as it provides some of the required functionality, such as managing components, resolving dependencies and distribution of components across different OSGi instances using R-OSGi [12]. The OSGi framework is also easily extendable using the hooks API.

The Execution Environment runs on top of the OSGi framework and R-OSGi, as shown in Figure 5. Each component is an OSGi bundle, whose providing interfaces are proxied by the EE. This way, the EE can easily monitor all method calls of the interface. The proxying of the interface also allows the EE to transparently migrate a component to a remote EE, redirecting all method calls through R-OSGi to the remote instance of the component.

On every node a Node Agent and a Cloudlet Agent are initially started, and all Execution Environments register with the Node Agent of the node they run on. When the node connects to a wireless network, other Cloudlet Agents are discovered using the Service Location Protocol (SLP) [18]. When another Cloudlet Agent is discovered in the LAN network, the Cloudlet Agent on one of the nodes (preferably the one with least resources) stops and its Node Agents register with the other Cloudlet Agent. This way an ad hoc cloudlet is formed. When the cloudlet has an uplink to the public internet, other cloudlets (i.e. an elastic cloudlet) can be registered by their IP address with the Cloudlet Agent, that will handle them as peers.

When an Execution Environment detects a constraint violation in one of the components, its Node Agent is signalled, which in its turn signals the Cloudlet Agent. The Cloudlet Agent can then instruct the EE to take actions to lower the load on the EE: configuration parameters of some components can be changed, or components can be outsourced to other nodes within the cloudlet or other known cloudlets.

The constraints and configuration parameters of a component are defined by the application developer using an XML description, as shown in listing 1. A typical constraint is the maximum execution time of a method. Configuration parameters are defined by their name, and can be configured by the EE using reflection.

```
<component name="Tracker">
  <service name="TrackerService">
    <method name="trackFrame">
      <constraint type="maxTime">50</constraint>
    </method>
  </service>
  <parameter name="MaxFeatures"
             type="uint"
             range="50..1000"
             default="200"/>
</component>
```

Listing 1: Example of a component description, defining the Tracker component with a constraint on the maximum execution time of the trackFrame method and a configurable parameter MaxFeatures.

The optimization goal of the Cloudlet Agent is to maximize the quality, while still meeting all the defined time constraints. We assume the quality of a component increases with the amount of resources used, or thus with the amount of (useful) work done. In the current implementation, the Cloudlet Agent only supports fixed predefined actions, but we plan to investigate different heuristics and learning techniques. Some preliminary work on algorithms for distributing components with multiple configuration options between a single mobile device and a server was presented in [19]. In this case, this has to be extended to a global optimization problem for multiple mobile devices and multiple servers.

We implemented the AR use case as a number of OSGi components as shown in Figure 3. To speedup the computation, most of the image processing is implemented in native C/C++ code, which is wrapped in OSGi components using the Java Native Interface (JNI). To be able to run the OSGi component on different platforms, native libraries are compiled for different architectures and the right library is selected and loaded at runtime by the OSGi framework. This way, the application components run both on x86 based machines running Linux and ARM based mobile devices running Android.

5. EVALUATION

With this evaluation we show that our cloudlet architecture with ad hoc cloudlets and a component management layer is beneficial for mobile rich media application that generate heavy load and have real-time constraints such as the augmented reality use case presented in Section 2.

We executed the AR use case on an ad hoc cloudlet consisting of a mobile device and a laptop connected via WiFi. The laptop is equiped with an Intel Core 2 Duo CPU clocked at 2.26GHz. As mobile device we use a HTC Desire, with a single core Qualcomm 1 GHz Scorpion CPU, and an LG Optimus 2x powered by a dual core Nvidia Tegra 2 CPU, also clocked at 1GHz. For both devices we run 3 fixed deployments: a first deployment with all components running on the mobile device, a second deployment with the Mapper and the ObjectRecognizer outsourced to the laptop, and a third deployment where also the real-time constrained components, the Tracker and the Relocalizer, are outsourced next to the Mapper and ObjectRecognizer. We also compare 2 configurations of the frame size of the frames captured by the VideoSource: 800x480 or 400x240 pixels.

The results for the HTC Desire are shown in Table 1, where the execution time (in ms) is given for tracking one frame (Tracker), for relocalizing one frame (Relocalizer), for initializing a 3D map (Map init), for refining the map with a new keyframe (Mapper) and for detecting known objects in a keyframe (Obj. Rec.). The most important action is the tracking of a frame, as this has to be done for each camera frame fetched, and will greatly determine the user experience. In order to offer a good quality to the end user, a framerate of 20 to 30 tracked frames per second is desired, which boils down to an execution time of less than 50 ms per frame for the Tracker.

The single core processor is not able to process 800x480 frames fast enough, having an average processing time of 147 ms to track a frame. Even when offloading to the cloudlet, the time to track one frame is still 85 ms. When configuring the VideoSource to fetch 400x240 frames, the time to track a frame is 87 ms. When outsourcing the Mapper and the ObjectRecognizer the processing time lowers to 54 ms on average, what is slightly more than the predefined constraint and the best configuration possible.

Notice that when also the Tracker and Relocalizer components (that have real-time constraints) are outsourced, the tracking time is still lower (68 ms) then running everything locally, resulting in a frame rate of 15 frames per second. In this case the time to track a frame is determined by the time to transfer the raw 400x240 frame from the mobile device to the laptop (60 ms), or thus by the bandwidth of the wireless access network. This is probably the best configuration on even lower end devices that do not have the CPU power to track frames locally. Also, as the biggest part of the time is spent on sending data, this configuration could be further optimized, e.g. by compressing the video frames.

Table 1: Execution time (in ms) on the HTC Desire for different operations for 2 configurations of the video source, and 3 deployments (everything local (1), Mapper and ObjectRecognizer outsourced (2) and Mapper, ObjectRecognizer, Relocalizer and Tracker outsourced (3)).

	800x480			400x240		
	1	2	3	1	2	3
Tracker	147	85	205	87	54	68
Relocalizer	95	69	2	21	16	1
Map init	2275	1783	130	1325	468	52
Mapper	3256	519	301	2627	455	150
Obj. Rec.	32835	1273	1442	19788	550	536

Table 2: Execution time (in ms) on the LG Optimus 2x for different operations for 2 configurations of the video source, and 3 deployments (everything local (1), Mapper and ObjectRecognizer outsourced (2) and Mapper, ObjectRecognizer, Relocalizer and Tracker outsourced (3)).

	800x480			400x240		
	1	2	3	1	2	3
Tracker	64	34	378	14	12	106
Relocalizer	21	21	2	3	3	1
Map init	2804	1124	176	429	390	82
Mapper	2470	519	396	547	305	156
Obj. Rec.	18682	1414	1510	6531	595	629

Also note that for the other CPU intensive tasks such as the map initialization and refinement, and the object recognition, remote execution gives a speedup of a factor 6 to 30. When comparing deployments 2 and 3, the processing time of the Mapper is lower in deployment 3, as the Tracker and Mapper are then colocated remotely, whereas in deployment 2 additional time is needed to transfer frames between the Tracker (on the mobile device) and the Mapper (on the laptop).

The LG Optimus 2x has a dual core processor, which leads to lower local processing times as shown in Table 2. For the Optimus the best configuration is processing 800x480 frames while outsourcing the Mapper and the ObjectRecognizer, in which case frames are tracked on average within 34 ms, which is equivalent with a framerate of about 30 frames per second. Although the 400x240 configuration has faster processing times for tracking a frame, the 800x480 configuration is chosen as this offers a better visual quality. Even with the better mobile processor, there still is a factor 2 to 12 speedup for the outsourced components.

Tables 1 and 2 show that indeed a component based approach is needed for real-time applications such as augmented reality. These results also show that for different devices a different configuration of the components is more beneficial than only outsourcing some components, and is an addition that greatly enhances the possibilities of the middleware to find a good deployment. When comparing the processing time of the Tracker in deployment 3 when the Tracker is outsourced, this processing time is higher for the LG Optimus compared to the HTC Desire. The time to transfer a camera frame from the device to the laptop accounts for this difference, as the actual time to analyse the frame on the laptop is the same in both cases. This shows that the network connectivity is highly variable, which can be taken into account in a fine grained component based partitioning.

To show the need for cloudlets in the LAN network, rather than outsourcing to a distant cloud, we executed the best configurations (highlighted in grey in Tables 1 and 2) with the cloudlet running in the 'distant' Amazon EC2 cloud. The results are shown in Table 3.

Table 3: Execution time (in ms) when offloading to the Amazon EC2 cloud. Although more CPU power is available in the cloud, data transfer slows down performance compared to the ad hoc cloudlet in Tables 1 and 2, due to the WAN latency.

	Desire 400x240 2	**Desire 400x240 3**	**Optimus 800x480 2**
Tracker	51	218	31
Relocalizer	10	1	20
Map init	1765	60	2766
Mapper	1417	157	1508
Obj. Rec.	521	598	1303

When outsourcing only the Mapper and the Object Recognizer (2), the execution times of the map initialization and the map refinement are higher than in the case of the ad hoc cloudlet, as those methods communicate with the mobile device. When also outsourcing the Tracker and Relocalizer component (3), the tracking time is over 200 ms due to the WAN latency, which does not come near the real-time requirements.

6. RELATED WORK

Offloading parts of applications from mobile devices to nearly discovered servers, called surrogates, has been proposed before under the term 'cyber foraging' [1]. The most recent cyber foraging frameworks are discussed below. For a more in depth discussion on cyber foraging systems we refer to the survey of Sharifi et al. [15].

Kristensen et al. [10] propose Scavenger, a cyber foraging system in Python, that offloads CPU intensive methods to enhance performance. A dual profiling scheduler is presented, using adaptive history-based profiles.

MAUI [5] outsources methods of Microsoft .Net applications in order to save energy. At initialization time MAUI measures the energy characteristics of the device, and during runtime the program and network characteristics are monitored. By solving an integer linear programming (ILP) problem a decision is made whether or not to offload a method.

Chun et al. present CloneCloud [3], where virtualized clones of the mobile device are executed in the cloud. In an offline profiling stage, different binaries of the application are generated, with special VM instructions 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.

Verbelen et al. [19] present a middleware framework in OSGi, that offloads OSGi components of an augmented reality application. Like MAUI, an ILP approach is used to determine the best partitioning. Also, when offloading is still not enough to make the application usable, the middleware will try to reduce the quality of some components and gracefully degrade the application.

Instead of offloading to a remote server or a cloud, one could also try to offload to other mobile devices in the vicinity [8]. This actually boils down to a special case of the cloudlet architecture, where an ad hoc cloudlet is formed between mobile devices, without connectivity to other cloudlets.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we present a cloudlet architecture that not only provides fixed infrastructure colocated with the WiFi access point, but also enables ad hoc discovery of devices in the vicinity to share resources among each other. Next to the provisioning of infrastructure, the proposed cloudlet architecture also provides a middleware framework to manage and distribute component based applications, with a focus on rich media applications such as augmented reality that have strict real-time requirements. Preliminary results show for an AR use case that a component based approach is indeed beneficial, and how the cloud is less suited for outsourcing components with real-time constraints.

As future work, whole new challenges have to be tackled with respect to deployment calculation and scheduling. In comparison with existing cyber foraging frameworks, a lot more decision options are generated by allowing different configurations of components. Instead of a single discovered surrogate, now multiple places for remote execution have to be considered, within the cloudlet or in other cloudlets. The hierarchical architecture also allows for decision taking for all users to reach a global optimum, instead of possibly conflicting local optima for each user individual.

Another area of interest is how to support application developers in creating such component based applications and defining their quality constraints. Developer tools are needed to easily split up an application in different components, in a (semi-) automatic way and with as less burden on the developer as possible.

8. ACKNOWLEDGMENTS

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

9. REFERENCES

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

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

[3] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages

301–314. ACM, 2011.

[4] B.-G. Chun and P. Maniatis. Dynamically partitioning applications between weak devices and clouds. In *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond*, MCS '10, pages 7:1–7:5, 2010.

[5] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and 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*, pages 49–62. ACM, 2010.

[6] Gartner Group. 2012 press releases. http://www.gartner.com/it/page.jsp?id=1924314.

[7] I. Giurgiu, O. Riva, D. Juric, I. Krivulev, and G. Alonso. Calling the cloud: enabling mobile phones as interfaces to cloud applications. In *Proceedings of the ACM/IFIP/USENIX 10th international conference on Middleware*, Middleware'09, pages 83–102, 2009.

[8] G. Huerta-Canepa and D. Lee. A virtual cloud computing provider for mobile devices. In *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond*, MCS '10, pages 6:1–6:5. ACM, 2010.

[9] G. Klein and D. Murray. Parallel tracking and mapping for small ar workspaces. In *Proceedings of the 2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality*, ISMAR '07, pages 1–10. IEEE Computer Society, 2007.

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

[11] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2).

[12] J. S. Rellermeyer, G. Alonso, and T. Roscoe. R-osgi: distributed applications through software modularization. In *Middleware '07: Proceedings of the International Conference on Middleware*, pages 1–20, New York, NY, USA, 2007. Springer-Verlag New York, Inc.

[13] M. Satyanarayanan. Mobile computing: the next decade. In *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond*, MCS '10, pages 5:1–5:6, 2010.

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

[15] M. Sharifi, S. Kafaie, and O. Kashefi. A survey and taxonomy of cyber foraging of mobile devices. *Communications Surveys Tutorials, IEEE*, (99):1–12, 2011.

[16] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 2002.

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

[18] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. Service Location Protocol. RFC 2165 (Proposed Standard), June 1997. Updated by RFCs 2608, 2609.

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