Resource Allocation for Cloud-Assisted Mobile Applications

Marvin Ferber, Thomas Rauber
Department of Computer Science
University of Bayreuth, Germany
{Marvin.Ferber,Rauber}@uni-bayreuth.de

Mario Henrique Cruz Torres, Tom Holvoet
Department of Computer Science
Katholieke Universiteit Leuven, Belgium
{MarioHenrique.CruzTorres, Tom.Holvoet}@cs.kuleuven.be

***Abstract*—Mobile devices such as netbooks, smart phones, and tablets have made computing ubiquitous. However, such battery powered devices often have limited computing power for the benefit of an extended runtime. Nevertheless, despite the reduced processing power, users expect to perform the same types of operations as they could do using their desktop or laptop computers.**

We address mobile devices's lack of computing power by leveraging cloud computing resources. We present a middleware that relocates computing-intensive parts of Java applications to cloud resources. Consequently, our middleware enables the execution of computing-intensive applications on mobile devices. We present a case study on which we adapt Sunflow, an open-source ray tracing application, to use our middleware and show the results obtained by deploying it on Amazon EC2. We show, via simulations, a cost analysis of using the different resource allocation strategies available on our solution.

I. INTRODUCTION

Ubiquitous computing has gained significant momentum over the past years, because prices of mobile devices like tablets and netbooks have decreased to an affordable level. The increased running time of such mobile devices, due to battery and power consumption improvements, has made computing feasible in nearly everywhere [1]. However, the computing capacity of such devices is restricted if compared to desktop computers or mainstream laptops [2]. This makes it unfeasible to execute compute-intensive applications on such devices. Thanks to the recent development in mobile communication technology, an internet connection is available in many outside places using WLAN or other communication technologies. The widespread communication infrastructure facilitates the integration of remote services into mobile applications. Moreover, due to the advent of cloud computing, computing power is publicly available on a pay-per-use model to any application that needs computational resources [3].

In this paper, we target the creation of predictable and scalable client/server applications for mobile devices. In contrast to dynamic offloading techniques like [4], we target compute-intensive Java applications that can be executed on a mobile device only when the computing intensive parts are relocated to a powerful server. We call such applications *Cloud-assisted mobile applications* (*CAM-apps*). *CAM-apps* are capable of performing computing-intensive calculations in a time and quality comparable to traditional desktop machines, opening a new set of possibilities for the creation of mobile applications. We exploit cloud computing techniques like virtualization and pay-per-use billing in order to enable on-demand provisioning of cloud servers. We present and evaluate a middleware based on cloud computing, more specifically based on Amazon Elastic Compute Cloud (EC2) [5]. As it is possible to predict the performance of cloud offerings [6], our middleware benefits from it to operate its remote compute services.

Our contribution is twofold.

1) We present a Java-based middleware that enables the creation of *CAM-apps*. The middleware manages all the deployment and execution of stateful services on the cloud provider, on behalf of the application developer. It also handles accounting and billing of these *CAM-apps*.

2) A special resource allocation strategy that guarantees a minimum service level quality in terms of processing speed for the *CAM-app*. The novelty of our approach is to assign each user request to a dedicated cloud resource. That way our middleware ensures a reasonable speedup that is necessary for our *CAM-app*.

In a case study, we present an adapted version of a real world application, Sunflow, on a mobile device supported by compute services deployed on Amazon EC2. We show the benefits and drawbacks of its execution as a *CAM-app*. We measure remote processing speed, cost, and energy savings and compare these results to an unmodified version of the application.

The rest of the article is structured as follows. We give an overview of our middleware in Section II. In Section III, we evaluate different resource allocation strategies regarding cost and scalability. We show an experimental evaluation of our middleware based on a case study, in Section IV. Finally, Section V discusses related works and Section VI presents our conclusions.

Table I
Amazon EC2 instance type characteristics (EU west)

	ECUs	Cores	RAM	Cost (USD)	Cost/ECU (US cent)
m1.small	1	1	1.7 GB	0.095	9.5
m1.large	4	2	7.5 GB	0.38	9.5
c1.medium	5	2	1.7 GB	0.19	3.8
m2.xlarge	6.5	2	17.1 GB	0.57	8.8
m1.xlarge	8	4	15.0 GB	0.76	9.5
m2.2xlarge	13	4	34.2 GB	1.14	8.8
c1.xlarge	20	8	7.0 GB	0.76	3.8
m2.4xlarge	26	8	68.4 GB	2.28	8.8

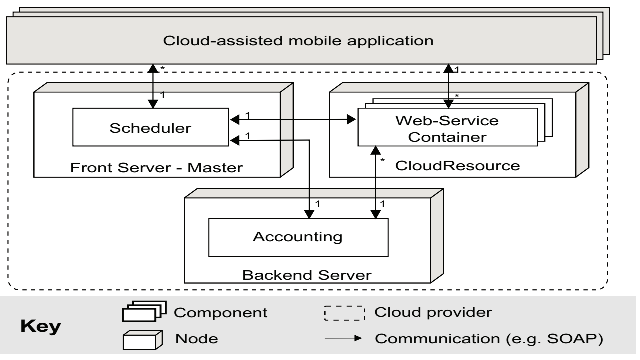

Fig. 1. Overview of the cloud middleware consisting of Master, Cloud Resources and an Accounting/Billing authority.

II. Overview of the Middleware

The cloud middleware presented in this section implements all server side features that are necessary to enable *cloud-assisted mobile applications* (*CAM-apps*). The basic features are:

- hosting the remote services,
- service management and allocation,
- accounting and billing.

The middleware is designed to support different network communication technologies for broad applicability and easy usage with existing as well as newly developed applications. To achieve this, a *remote service* represents a stateful resource that is used as a *remote object* (*ro*). A *ro* instance is created (constructor) on request, and can be used until it is explicitly destroyed (destructor). As this reflects the life cycle of Java objects, they can easily be used as a *ro* in our middleware. A wrapper class that implements the network communication, e. g. SOAP stack, must be provided. The communication behavior used in our middleware has already been proposed for SOAP Web services in [7]. The idea of this approach is to use a remote object id (session key) that identifies a Java object instance behind a (stateless) Web service interface in order to enable *Remote Object over Web Services*. We apply the session key mechanism to also support REST, RMI and CORBA in our middleware.

The middleware consists of a fixed *master* server, a fixed server for *accounting/billing*, and an adjustable set of *cloud resources* that contain instances of the actual *remote services* (*ro′s*). Fig. 1 gives an overview of our cloud-based middleware. In our middleware, a *cloud resource* is a complete virtual machine (*vm*) which runs the *ro*. A *client* requests a *ro* from the *master* server and can specify a desired service quality (e. g., *vm* instance type), possibly adjusting the Service Level Agreement (SLA) dynamically at runtime. Prior to delivery, the *client* is authenticated and a session key is created. After delivery (`create`), all subsequent communication (`use, destroy`) is performed directly between the *client* and the *vm* in a point-to-point manner using the session key. A more detailed example of the life-cycle of such a *ro* is given in Fig. 6 in our case study.

All communication is logged into a database by the *vm*, using the session key as identifier. This information can be used later for billing. Furthermore, all communication is encrypted by default using a certificate based SSL encryption. The encryption is used to protect user data and to prevent malicious manipulation, e. g. stealing of the session key. Using a session key is also a key concept in online banking web portals. However, they use a lease time to destroy a session key automatically after a certain idle period. This can also be implemented with our approach but may not be desired by all applications because of, e. g., long lasting computations.

The prototypical middleware implementation is based on Amazon EC2. Amazon provides different virtual machine types with 1 to 8 cores and up to 64 GB of main memory. Table I shows the available instance types from Amazon EC2. Each EC2 instance can run different Amazon Machine Images (AMI). For each *ro* that should be available in the middleware, a 32 bit and a 64 bit AMI is created. The *master* server knows about all running EC2 instances, their booted images as well as their current status.

As mentioned before, the session key reflects the life cycle of the *ro*, which consists of: `create, use`, and `destroy`. The session key solves multiple problems like authentication, addressing and billing. As the session key is independent from the underlying network technology, it also provides a concept of fault tolerance against network failure. If the clients network link goes down, the user can switch to another access point or network interface, e. g. from WLAN to 3G/4G, and continue the computation afterwards, which is helpful for possibly unstable wireless connections. In contrast to other remote computing techniques, as `Virtual Network Computing (VNC)` or `X11`, we aim at reducing network traffic by only transferring the necessary data, avoiding transfering the whole graphical user interface (GUI).

Although our middleware is independent from the underlying network communication technology, specific technology attributes need to be taken into account when creating remote cloud services, e. g., firewall restrictions, parameter encoding, or interoperability with different programming languages. A comparison of the supported technologies can be found in [8] and [9]. We show the influence of different network technologies in our case study, in Section IV.

The invocation behavior on the *ro* needs to comply to common remote object behavior. Since we only use reliable and encrypted TCP connections, a network timeout may appear while processing large data sets remotely. On the contrary, transferring large data sets maximizes communication throughput. As a result, invocation frequency is a trade-off between message size and the possibility of a network timeout. However, even with a high invocation frequency, the communication overhead can be decreased by overlapping communication and computation (parallel method invocations). Overlapping implies the presence of more than one CPU core to be effective. Besides `m1.small`, all EC2 instance types from Table I have at least 2 cores.

III. Cloud Resource Allocation Simulations

To do remote computation effectively, each user request should be assigned to a dedicated *vm*. Moreover, different resource allocation strategies have different impacts on cost and service level quality of the *remote services* (*ro*). We have performed simulations to analyze the characteristics of two different allocation strategies.

The simulation model is described in Table II. Two service quality attributes have been taken into account: (i) allocation time (Δt_{alloc}) and (ii) processing time overhead (Δt_{shared}). We implemented the aforementioned exclusive resource allocation strategy that guarantees a fixed processing speed and a traditional strategy that allocates resources immediately by multiplexing requests.

A. Simulation model

The performed discrete event simulation is represented by a 5-tuple $Sim(VMs, ROs, S_{init}, S_{final}, F_{alloc})$. Each run simulates one day. A cloud resource (*vm*) is described by its creation time (t_{create}), the duration of its startup phase ($\Delta t_{startup}$), its killing time (t_{kill}), and its current occupation (num_{ro}). We do not target a specific EC2 instance type for our simulation, although the simulation is based on the EC2 environment characteristics, e. g., hourly billing period. A remote object (*ro*) is described by the time a client requests the instantiation of the *ro* ($t_{request}$), the time it is really started (t_{start}), its processing time (Δt_{work}), and the time when all work has been processed (t_{finish}). Δt_{work} is the duration of processing all work of the *ro* while using an exclusive *vm* throughout the whole computation. If the *ro* needs to share a *vm* with another *ro*, then $\Delta t_{shared} > 0$.

Table II
Description of the simulation model.

Definition	**Description**
$Sim(VMs, ROs, S_{init}, S_{final}, F_{alloc})$	description of the simulation
S_{init}, S_{final}	initial/final simulation state
t_{sim}	current simulation time point
$num_{standby} \geq 1$	$\lvert VM_{free}\rvert$ should be $num_{standby}$
$vm(t_{create}, \Delta t_{startup}, t_{kill}, num_{ro})$	description of a cloud resource
$VMs = \{vm_0, \ldots, vm_x \lvert x \in \mathbb{N}\}$	set of cloud resores
$load(vm)$	returns num_{ro} at t_{sim}
$free(vm) = true$	if $num_{ro} == 0$
$ready(vm) = true$	if $(t_{create} + \Delta t_{startup}) \leq t_{sim}$
$VM_{free} = \{vm \in VMs \lvert$ $free(vm) \wedge ready(vm)\}$	set of free cloud resources
$ro(t_{request}, t_{start}, \Delta t_{work}, t_{finish})$	description of a remote object
$ROs = \{ro_0, \ldots, ro_x \lvert x \in \mathbb{N}\}$	set of remote objects
Δt_{alloc}	$t_{start} - t_{request}$
Δt_{shared}	$t_{finish} - t_{start} - \Delta t_{work}$
$\Delta t_{overhead}$	$\Delta t_{alloc} + \Delta t_{shared}$
$F_{alloc}(ro)$	allocates a vm for a ro request
$F_{assign}(vm, ro)$	assigns a ro to a specfic vm
$F_{adjust}(ROs, VMs)$	adjust $num_{standby}$ of VMs

At the beginning of each simulation, the initial VM set and the final state are empty ($VMs = S_{final} = \emptyset$). Each simulation starts with restoring the initial state S_{init} and setup of the set of *ROs* from corresponding input files. In the simulation, new $ro's$ are assigned on request $t_{request}$ to a *vm* using the provided allocation function F_{alloc}. The remaining work time Δt_{work} of a running *ro* is decreased each second depending on the actual occupation of the *vm* they are running on. For $ro's$ that run on a *vm* exclusively, the remaining work time is decreased by 1 s. For $ro's$ that share a *vm* with other $ro's$, the work time is decreased by $\frac{1}{load(vm)}$ s only. Finished $ro's$ ($\Delta t_{work} == 0$) are removed from the *vm* they were running on. Furthermore, $num_{standby}$ is adjusted by starting new $vm's$ or by killing unnecessary $vm's$ periodically. Unnecessary $vm's$ are kept running as possible standby instances and are killed right before a new payment period starts (hourly billing). After a simulation run, VMs contains all $vm's$ that have been used during the simulation including their t_{create} and t_{kill}. S_{final} contains all $vm's$ and $ro's$ that were running at the end of the simulation. S_{final} can be used as S_{init} for the simulation run of the following day. Thus, it is possible to perform a continuous simulation. The t_{start} and t_{finish} of each $ro \in ROs$ are updated according to the simulation run.

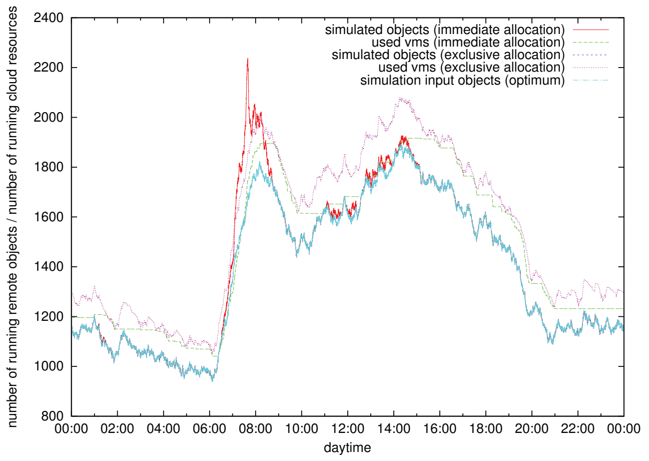

Fig. 2. Distribution of 100,000 simulated requests and respective *vm* allocation using different resource allocation strategies

B. Generating simulation input

Simulation input was generated by randomly distributing a number of requests between 00:00 and 24:00 hours. Each request represents the beginning of the usage of a remote service (ro) for a period of time (Δt_{work}). To make the distribution of requests more realistic, we use a distribution function based on [10]. Using this distribution, which takes daytime into account. it is more likely that a request is sent between 8:00 and 18:00 than at night. A sample distribution of 100,000 requests (Δt_{work}=20 min) can be seen in Fig. 2.

Although the average vm startup times found by [11] are lower, we measured a $\Delta t_{startup}$ of around 150 s for all instance types. This may be due to a different Amazon EC2 region (e. g., US or Asia), compared to our benchmarks in the EU West region. Unfortunately, the region is not mentioned in [11]. We fixed $\Delta t_{startup}$ to 150 s to get comparable results.

C. Resource allocation strategy

As stated above the objective of the resource allocation strategy is to always have enough running standby instances to serve all incoming requests immediately and exclusively. According to cost constraints, the number of standby instances ($num_{standby}$) needs to be adjusted to the number of running requests. As a trade-off between cost and an insufficient number of free $vm's$, we adjust $num_{standby}$ to 10% of the number of currently running ro ($num_{standby} = |RO_{current}| \cdot 0.1$ with $RO_{current} = \{ro \in ROs \,|\, ro.t_{start} \leq t_{sim} < ro.t_{finish}\}$). Our allocation strategy guarantees that at least 1 vm is always running.

However, the allocation strategy cannot guarantee that a free standby instance is available if there are several requests in a short period of time. Taking all the requirements into account, we created two different resource allocation strategies F_{alloc}.

The first strategy never allocates more than one remote service (ro) to a single vm. Accordingly, in the worst case a client needs to wait $\Delta t_{startup}$ until a new vm is started up to serve the request. The advantage of this strategy is that a ro can always use the full capacity of a vm and thus $\Delta t_{shared} = 0$. As a result, the maximum overhead $MAX(\Delta t_{overhead}) = \Delta t_{startup}$. The corresponding allocation strategy function $F_{alloc\,exclusive}$ is shown in Algorithm 1.

Algorithm 1 F_{alloc} vm exclusively

```
Require: $ro$ {place a remote object on a $vm$}
  if $|VM_{free}| < 1$ then
    $vm_n = createnewVM()$
    $VMs = VMs \cup \{vm_n\}$
    while $\neg ready(vm_n)$ do
      wait(1s)
    end while
    $F_{assign}(vm_n, ro)$
  else
    $F_{assign}(vm_x, ro)$ with $vm_x \in VM_{free}$
    $VM_{free} = VM_{free} \setminus vm_x$
  end if
  $F_{adjust}(ROs, VMs)$
```

The second allocation strategy is designed to return a vm on request immediately ($\Delta t_{alloc} = 0$) to reduce the waiting time for users. If there are no available standby instances ($|VM_{free}| = 0$), the request is multiplexed to the least occupied vm. Unfortunately, this strategy may increase Δt_{shared} of the ro significantly, due to overloading a vm. In contrast to $F_{alloc\,exclusive}$, no upper bound for $\Delta t_{overhead}$ can be guaranteed. This strategy is summarized in Algorithm 2.

Algorithm 2 F_{alloc} vm immediately

```
Require: $ro$ {place a remote object on a $vm$}
  if $|VM_{free}| < 1$ then
    $F_{assign}(vm_n, ro)$ with $vm_n = \underset{vm_n \in VMs}{\operatorname{argmin}} \big(load(vm_n)\big)$
  else
    $F_{assign}(vm_x, ro)$ with $vm_x \in VM_{free}$
    $VM_{free} = VM_{free} \setminus vm_x$
  end if
  $F_{adjust}(ROs, VMs)$
```

D. Simulation results

We have conducted simulations starting with 100 until 1,000,000 requests per day ($\frac{\#_{req}}{day}$). We have also varied Δt_{work} of the $ro's$ from 2 s up to 20 min. This setup not only reflects different workloads, it can also be interpreted as different vm speeds. A reduced Δt_{work} can be due to less work or due to a faster vm. Each simulation has been performed to simulate several days using S_{final} of one simulated day as S_{init} of the following simulated day, always using the same simulation input until the results stabilized. Here, we show the average results over 10 stabilized simulation runs.

Table III
vm DEMANDS FOR THE DIFFERENT ALLOCATION STRATEGIES (NUMBER OF SIMULTANEOUSLY RUNNING $vm's$)

request length	2					10					60					300					1200				
requests per day	100	1000	1E+4	1E+5	1E+6	100	1000	1E+4	1E+5	1E+6	100	1000	1E+4	1E+5	1E+6	100	1000	1E+4	1E+5	1E+6	100	1000	1E+4	1E+5	1E+6
immediate allocation (max)	3	4	5	11	80	3	4	9	42	273	3	6	29	162	1022	4	12	67	530	4902	6	30	223	1931	20157
immediate allocation (avg)	1,81	2,52	3,83	7,69	45,22	1,73	2,81	5,89	22,76	154,19	1,80	3,57	14,62	97,21	752,47	2,07	6,83	44,27	376,24	3674,83	3,00	17,84	152,57	1481,46	14732,23
exclusive allocation (max)	2	3	5	7	182	3	4	11	49	199	3	8	25	138	1091	4	15	66	547	5311	8	31	235	2079	20768
exclusive allocation (avg)	1,67	2,15	3,40	6,57	43,28	1,65	2,60	5,73	22,72	151,09	1,68	3,85	15,54	91,39	797,48	2,11	7,58	46,95	398,87	3852,73	3,35	19,72	165,44	1546,87	15330,45

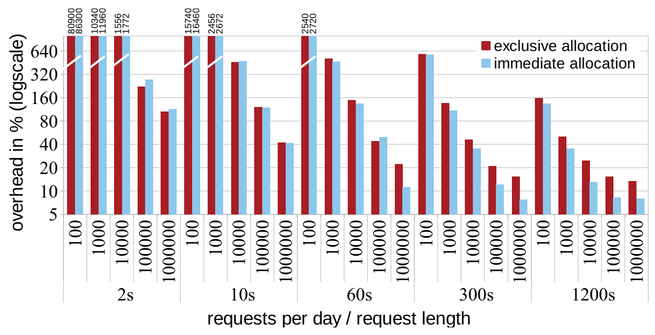

Fig. 3. overall vm idle time overhead

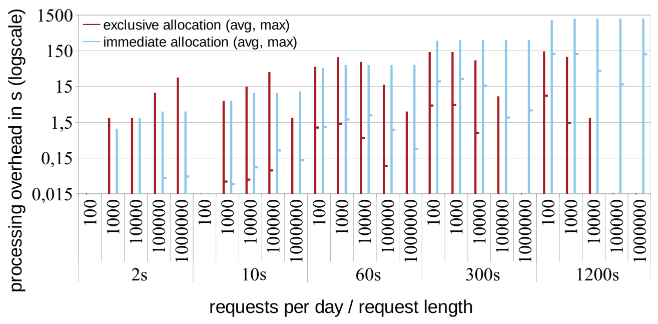

Fig. 4. $\Delta t_{overhead}$ for simulated $ro's$ (average and max values)

Fig. 2 shows simulation results for 100,000 $\frac{\#_{req}}{day}$ and a Δt_{work} of 20 min. The diagram shows the sum of currently running remote services (ro) and virtual machines (vm) over time, for both resource allocation strategies in comparison to the simulation input data. The spacing between the input data in the graphs of ro and vm reflect the overhead to the optimal allocation strategy. Fig. 2 shows that the ro graph for the exclusive allocation almost covers the input graph, but this strategy also uses more $vm's$ than the immediate allocation strategy. On the other hand, the ro graph for the immediate allocation strategy is often above the simulation input graph, which indicates an overhead $\Delta t_{overhead}$. This can especially be seen when the number of ro requests increases in a short period of time, e. g., between 6:00 and 10:00. This is due to the fact that $F_{alloc\,exclusive}$ not only adjusts the number of $vm's$ periodically, according to the number of $ro's$, it always creates a new vm on request if there are no free $vm's$ left. As a result $F_{alloc\,exclusive}$ can react more quickly when the number of requests increases in a short period of time.

Table III shows the average and maximum number of simultaneously running $vm's$ used by the different resource allocation strategies to serve different $\frac{\#_{req}}{day}$ and Δt_{work}. It can be seen that both strategies do have similar demands. Fig. 3 shows that the vm over-provisioning in terms of unused $vm's$ idle time is the lowest with high number of $\frac{\#_{req}}{day}$ and long Δt_{work}. The overhead is also a measure of cost efficiency, because a low overhead means a high occupancy ratio, which results into a lower price.

The advantage of the exclusive allocation strategy is shown in Fig. 4. The overhead $\Delta t_{overhead}$ is never above $\Delta t_{startup}$ (150 s in our case). This means that although a user may need to wait a bit until a ro is started on request, it can then use the full vm computing power. This upper bound, aforementioned, fast standby instance adjusting mechanism ensure a predictable overhead for this strategy. For a high number of $\frac{\#_{req}}{day}$ and long Δt_{work} the overhead decreases, because of a high $num_{standby}$ and high vm reuse probability. As Fig. 2 shows, the immediate allocation strategy has problems to adjust the number of standby instances when the number of request increases very quickly. A high $\Delta t_{overhead}$ is the result of such strategy. This can also be seen at the high max values in Fig. 4. However, the average overhead values are much lower than the max values for both strategies.

E. Traffic Cost

All traffic to/from Amazon datacenters implies an indirect cost because communication time means a virtual machine instance running time on Amazon EC2. Additionally, Amazon charges download traffic in different steps. For our traffic cost estimations, we assume the highest cost of 0.12$ per GB, which is a pessimistic assumption. Fig. 5 shows the cost of traffic relatively to the computation for different simulated scenarios. For longer request lengths we assumed an increasing download traffic. The diagram shows results from 0,2 up to 60 MB. As traffic costs per request are always constant, the diagram also reflects the overhead cost in terms of unused $vm's$ when traffic costs are relatively low. This can be recognized with a decreasing number of requests per day (see Fig. 3) as well as with an increasing performance of the EC2 instance type. As less powerful instances are cheaper, the wasted money through over provisioning is lower and thus traffic cost

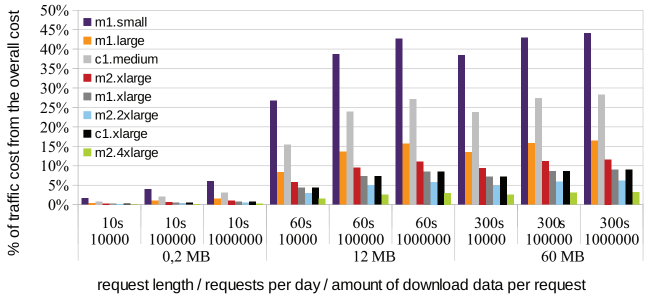

Fig. 5. Percentage of traffic cost from overall cost.

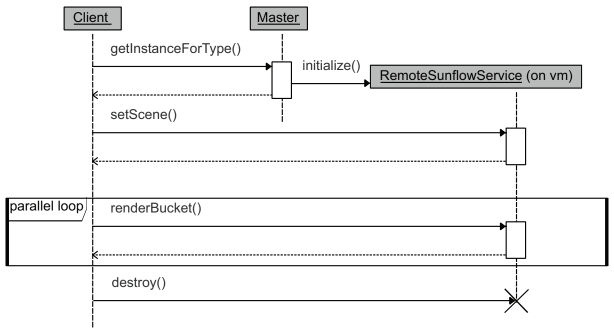

Fig. 6. Sequence of invocations on the remote Sunflow service.

has a greater influence. In conclusion it can be said that traffic cost is often low compared to computation cost in our simulations. We left out the cost for storage here since it was only 0.0024 ¢ for each computation, which is negligible in this case.

IV. Case Study

We have adapted a desktop application called Sunflow, in order to demonstrate the use of our prototypical cloud-based middleware. Sunflow[1] is an open source ray tracing application that can render 3D scenes by making use of multicore processors. Sunflow is part of the SPECjvm2008 benchmark suite [12] and has already been used, in [13], for distributed rendering using applets. Our transformed cloud-assisted Sunflow version has been investigated regarding its performance on different Amazon EC2 instances compared to a purely local execution. All measurements have been conducted in the EU west region of Amazons EC2 environment.

Sunflow consists of a GUI, a scene parser and different rendering engines. Sunflow can be used for rendering 3D scenes that have been created using Open Source 3D creation tools such as Blender[2]. For our investigations, we have used an example scene from the Sunflow example scene pack named `aliens_shiny.sc`. We have divided Sunflow into a client and a server part. We have extracted the multicore-capable bucket renderer and the scene parser and have refactored a self-contained remote service (*ro*) that can be executed on a cloud resource (*vm*). Only the GUI is left on client side. The resulting communication pattern between the Sunflow client and the Sunflow *remote service* (*ro*) is shown in Fig. 6. After having obtained a *vm* from the Master (`getInstanceForType()`), the scene file is sent to the *ro* and parsed. As a return value of `setScene()`, the *ro* informs the client about the number of cores the *vm* is able to use. The client then starts the simultaneous rendering by requesting single buckets (`renderBucket()`), according to the number of server cores returned. The request overlapping hides communication time and increases server occupation in order to use all available server cores efficiently. After all buckets have been rendered, the client destroys the *ro* (`destroy()`).

We have used an Asus eeePC netbook with an Intel Atom N270 CPU (1 core 1,6 GHz HT), 1 GB RAM, Wi-Fi 802.11n and Windows XP as the mobile device. We have also used a regular laptop with an Intel Core2 Duo (2.8 GHz 2 cores), 8 GB RAM, and openSUSE Linux 11.3 64 bit to compare the achieved performance of the *CAM-app*. We used the Sun/Oracle Java virtual machine (JVM) 1.6 on all computers. The middleware components of the `Master` and `Billing` service were executed on dedicated EC2 `m1.small` instances. Besides standard Java RMI, we used RESTeasy[3], JacORB[4] and Axis2[5] to integrate REST, CORBA and SOAP communication technologies in our middleware.

In a first test, we have determined a suitable bucket size that assures a good balance between message size and request frequency. Runtime tests have shown that a bucket size of 32x32 pixels results into an average message size of around 3 kb and an average computation time of around 1 s for each bucket. The bucket size chosen leads to 300 buckets and thus 303 requests including creation, scene initialization and destruction of the remote service. Fig. 7 (middle) shows the overall network traffic size that has to be transferred using different technologies and a bucket size of 32x32 pixels. Because of the (partially) textual encoding, SOAP and REST create larger messages than RMI and CORBA, that transfer binary data. We used the Wireshark[6] network analyzer to measure the network traffic produced using these different communication protocols.

Fig. 7 (left) shows the overall computation time to

[1]http://sunflow.sourceforge.net/
[2]http://www.blender.org/
[3]http://www.jboss.org/resteasy
[4]http://www.jacorb.org/
[5]http://ws.apache.org/axis2/
[6]http://www.wireshark.org/

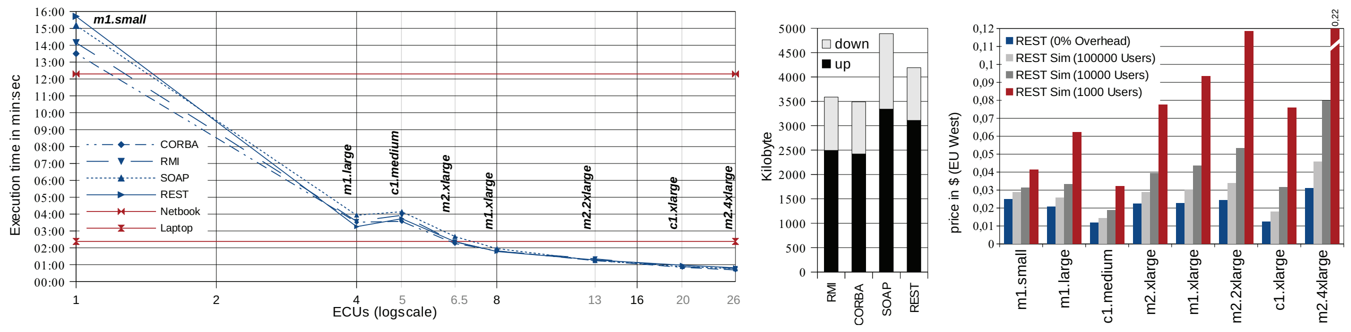

Fig. 7. Left: Execution time of the *CAM-app* Sunflow on different EC2 instance types compared to local execution. Middle: Traffic demands for different communication technologies. Right: Cost per computation for REST for 100% utilization and a real utilization based on simulation.

complete the sample scene, including construction and destruction of the *ro*. Each point in the diagram reflects the average value of three tests on a specific EC2 instance using a specific communication technology. The diagram shows that the overall execution time decreases according to the number of ECUs available on the EC2 instances. The performance of the cloud-assisted version of Sunflow is much better than the pure netbook version (maximum speed up of 17 for 26 ECUs using CORBA) and is comparable to the laptop version when using more than 6 ECUs. This means that the remote Sunflow service efficiently uses the available cloud resources. Thereby, the influence of the network technology chosen is not significant for this case study. This is also due to the partially overlapped communication and computation.

Comparing remote and local execution directly, the power consumption on the netbook is reduced by ~6%, but ~50% saving could be achieved at the laptop. Taking into account the time saving due to an accelerated remote execution, the overall power consumption of the netbook is decreased up to 16% (up to 52% on the laptop). The low energy savings on the netbook are due to a small power consumption gap of only ~1,5 W between idle and full cpu usage (~30 W on the laptop).

The cost per computation for the *CAM-app* version on EC2 instances is shown in Fig. 7 (right). It shows the simulation results using the measured Δt_{work} times from Fig. 7 (left) and our exclusive *vm* allocation strategy. It can be seen that the `c1` instances are the most efficient in our example. As a result, a user should only decide between `c1.medium`, `c1.xlarge` and `m2.4xlarge` because all other instances provide less performance for at least the same money.

V. Related Work

Recently Cloud Computing has been used to improve the execution of applications on mobile devices regarding processing speed and power consumption.

Different aspects of the development of mobile applications using RESTful Web services are described in [14]. The work of [4] targets power consumption optimization by code offloading to the cloud. The work of [15] presents a web-services based Java classloader called (WSBCL). WSBCL is able to load Java classes stored on remote computers without the need of adding any other software servers to these computers. Similar to our work, it transparently handles the Java class loading process of classes hosted on different hosts over the network. In contrast to our work, these articles mainly describe client-side aspects of cloud-based client/server applications.

One aspect of our middleware deals with the environment that mobile applications can use to leverage the processing power of cloud computing resources. Middleware for Software-as-a-Service (SaaS) applications such as Microsoft Azure or Googles App engine already exist [16], usually providing a Platform-as-a-Service (PaaS) in contrast to our solution. Our middleware can be seen as a blueprint of how to provide a SaaS on top of Amazon EC2, completely hiding the PaaS layer from the application developer.

Regarding resource allocation strategies we can cite [17], that proposes a decentralized approach to dynamically adapt cloud resource's usage. The main difference between this approach and our scheduling strategies is the focus on guaranteeing the contracted SLA's on time of high demand for the deployed services, while our approach tries to guarantee the SLA conformance for every single request. Another difference is that we deal with stateful services instead of stateless ones. Resource allocation in utility computing is furthermore targeted in [18]. Like in our case study, remote image rendering is targeted in this work. However, the therein presented resource allocation strategy pursues overall computation cost optimization in contrast to our work that that aims at service quality optimization on the client side and cost optimization at the cloud side.

An approach to optimizing resource usage and guaranteeing SLA's is presented in [19], which describes a game theoretical model to devise strategies for SaaS providers to schedule new machines on cloud providers. Our model focuses on minimizing the total time for completing a job on the cloud, while avoiding having too many unused resources. We believe we can also improve our resource allocation approach by incorporating features from the model proposed in [19]. [20] presents a report on the performance of Amazon EC2 resources. Our work evaluates the impact of different cloud resources on the performance of a particular application.

VI. Conclusion

We have presented a middleware that enables the development of *cloud-assisted mobile Java applications* (*CAM-apps*). We have also presented a resource allocation strategy aimed at the Amazon EC2 environment. Our strategy ensures a reasonable speedup of such *CAM-apps*. We compared our cloud resource allocation strategy with a traditional approach, regarding cost and performance. Our allocation strategy provides a predictable processing speed, making it feasible to guarantee specific service-level agreements. We did a case study using an open source ray-tracing application, called Sunflow, showing the benefits of *CAM-app*'s. We also show the different qualities regarding performance and pricing in the manner of a SaaS using our middleware. Our case study shows that a *CAM-app* executing on a netbook can reach the performance characteristics of the same application executing on a regular laptop. We also evaluate the costs of using the cloud, particularly Amazon EC2, to assist *CAM-app*'s. Finally, we show a feasible solution to create *CAM-app*'s, leveraging the benefits of cloud computing while, at the same time, hiding its complexities from application developers.

References

[1] S. Poslad, *Ubiquitous computing: smart devices, environments and interactions*. John Wiley & Sons Inc, 2009.

[2] R. C. Kennedy, "Clocking windows netbook performance," JUNE 2009. [Online]. Available: http://podcasts.infoworld.com/d/hardware/clocking-windows-netbook-performance-883

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, 2009.

[4] K. Kumar and Y.-H. Lu, "Cloud Computing for Mobile Users: Can Offloading Computation Save Energy?" *Computer*, vol. 43, no. 4, pp. 51 –56, april 2010.

[5] "Amazon Elastic Compute Cloud (Amazon EC2)." [Online]. Available: http://aws.amazon.com/ec2/

[6] A. Lenk, M. Menzel, J. Lipsky, S. Tai, and P. Offermann, "What Are You Paying For? Performance Benchmarking for Infrastructure-as-a-Service Offerings," in *IEEE Int. Conf. on Cloud Computing (CLOUD 2011)*. IEEE, 2011, pp. 484–491.

[7] M. Ferber, T. Rauber, and S. Hunold, "Combining Object-Oriented Design and SOA with Remote Objects over Web Services," in *Proc. of the 8th European Conf. on Web Services (ECOWS 2010)*, 2010, pp. 83–90.

[8] N. Gray, "Performance of Java Middleware-Java RMI, JAXRPC, and CORBA," in *Proc. of the 6th Australasian Workshop on Software and System Architectures (AWSA 2005)*, 2005, pp. 31–39.

[9] N. Nurseitov, M. Paulson, R. Reynolds, and C. Izurieta, "Comparison of JSON and XML Data Interchange Formats: A Case Study," in *Proc. of the ISCA 22nd Int. Conf. on Computer Applications in Industry and Engineering (CAINE 2009)*, 2009, pp. 157–162.

[10] P. Saripalli, G. Kiran, R. R. Shankar, H. Narware, and N. Bindal, "Load Prediction and Hot Spot Detection Models for Autonomic Cloud Computing," in *Proc. of the Int. Conf. on Utility and Cloud Computing (UCC 2011)*. IEEE, 2011, pp. 397–402.

[11] A. Iosup, S. Ostermann, M. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 6, pp. 931 –945, june 2011.

[12] H. Oi, "A Preliminary Workload Analysis of SPEC jvm2008," in *Proc. of the Int. Conf. on Computer Engineering and Technology*. IEEE, 2009, pp. 13–19.

[13] A. McMahon and V. Milenkovic, "Rendering Animations with Distributed Applets," in *Proc. of the Int. Conf. on Computer Graphics and Virtual Reality (CGVR2009)*, 2009, pp. 136–142.

[14] J. H. Christensen, "Using RESTful Web-Services and Cloud Computing to Create Next Generation Mobile Applications," in *Proc. of the 24th Conf. on Object-Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, 2009, pp. 627–634.

[15] F. Teixeira, M. Santana, R. Santana, S. Bruschi, and J. Estrella, "WSBCL: Web Services Based Classloader," in *20th IEEE Int. Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2011)*. IEEE, 2011, pp. 128–133.

[16] R. Buyya, C. Yeo, and S. Venugopal, "Market-Oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities," in *Proc. of the 10th Int. Conf. on High Performance Computing and Communications (HPCC2008)*. IEEE, 2008, pp. 5–13.

[17] N. Bonvin, T. Papaioannou, and K. Aberer, "Autonomic SLA-Driven Provisioning for Cloud Applications," in *11th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing (CCGrid 2011)*, may 2011, pp. 434 –443.

[18] J. a. N. Silva, L. Veiga, and P. Ferreira, "Heuristic for Resources Allocation on Utility Computing Infrastructures," in *Proc. of the 6th Int. Workshop on Middleware for Grid Computing (MGC 2008)*. ACM, 2008, pp. 1–6.

[19] D. Ardagna, B. Panicucci, and M. Passacantando, "A Game Theoretic Formulation of the Service Provisioning Problem in Cloud Systems," in *Proc. of the 20th Int. Conf. on World Wide Web (WWW 2011)*. ACM, 2011, pp. 177–186.

[20] J. Dejun, G. Pierre, and C. Chi, "EC2 Performance Analysis for Resource Provisioning of Service-Oriented Applications," in *Proc. of the 3rd Workshop on Non-Functional Properties and SLA Management in Service-Oriented Computing*. Springer, 2009, pp. 197–207.