Multi-Cloud Provisioning and Load Distribution for Three-Tier Applications

NIKOLAY GROZEV and RAJKUMAR BUYYA, University of Melbourne, Australia

Cloud data centers are becoming the preferred deployment environment for a wide range of business applications because they provide many benefits compared to private in-house infrastructure. However, the traditional approach of using a single cloud has several limitations in terms of availability, avoiding vendor lock-in, and providing legislation-compliant services with suitable Quality of Experience (QoE) to users worldwide. One way for cloud clients to mitigate these issues is to use multiple clouds (i.e., a Multi-Cloud). In this article, we introduce an approach for deploying three-tier applications across multiple clouds in order to satisfy their key nonfunctional requirements. We propose adaptive, dynamic, and reactive resource provisioning and load distribution algorithms that heuristically optimize overall cost and response delays without violating essential legislative and regulatory requirements. Our simulation with realistic workload, network, and cloud characteristics shows that our method improves the state of the art in terms of availability, regulatory compliance, and QoE with acceptable sacrifice in cost and latency.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architectures; C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms: Performance, Legal Aspects, Experimentation

Additional Key Words and Phrases: Cloud computing, Multi-Cloud, three-tier applications, autoscaling, load balancing

ACM Reference Format:
Nikolay Grozev and Rajkumar Buyya. 2014. Multi-Cloud provisioning and load distribution for three-tier applications. ACM Trans. Autonom. Adapt. Syst. 9, 3, Article 13 (October 2014), 21 pages.
DOI: http://dx.doi.org/10.1145/2662112

1. INTRODUCTION

Cloud computing is a disruptive IT paradigm that changes the way businesses operate. Instead of owning, maintaining, and administering their own infrastructure, businesses can now dynamically rent resources on demand just as they need them [Buyya et al. 2009; Mell and Grance 2011]. This allows them to avoid upfront investments in infrastructure that may not fit their dynamic needs at all times, being either under- or overutilized. Moreover, enterprises can now eliminate activities like infrastructure maintenance and administration and focus on their core business operations.

The standard model of consuming a cloud service is when a client uses resources within a single cloud. However, this poses several challenges for cloud clients. First, a data center outage can leave clients without access to resources, as exemplified by the outages of several major vendors [Amazon 2014e, 2014f; Google 2014; Microsoft 2014].

Authors' addresses: N. Grozev and R. Buyya are with the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, Department of Computing and Information Systems, The University of Melbourne, Parkville, VIC 3010, Australia; email: ngrozev@student.unimelb.edu.au; rbuyya@unimelb.edu.au.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2014 ACM 1556-4665/2014/10-ART13 $15.00
DOI: http://dx.doi.org/10.1145/2662112

And, as per Berkeley's report, cloud service unavailability is the greatest inhibitor to cloud adoption [Armbrust et al. 2009]. Second, interactive online applications (e.g., three-tier systems) usually have network latency constraints. A single data center cannot serve users distributed worldwide with adequate latency. Last, many businesses that operate across national boundaries need to comply with different regulations in terms of privacy, security, and data location. This is of special importance for applications dealing with sensitive data (e.g., in the banking or e-health domains), and businesses are often required to use data centers within a given territorial jurisdiction when serving some customers [Bowen 2011]. It is unlikely that a single data center will comply with the constraints of all targeted jurisdictions.

To overcome these issues, researchers and practitioners have envisioned the usage of multiple clouds. A Multi-Cloud is a type of Inter-Cloud, in which clients utilize multiple clouds without relying on any interoperability functionalities implemented by the providers [Ferrer et al. 2012; Petcu 2013; Grozev and Buyya 2012]. Application deployment across clouds has recently attracted interest and resulted in the emergence of at least 20 projects facilitating cross-cloud deployment [Grozev and Buyya 2012]. Case studies by IBM [2013] and e-Bay [2014] have demonstrated how three-tier interactive applications can utilize multiple data centers to provide better availability and customer Quality of Experience (QoE) and to quickly adapt to changes in demand.

Unfortunately, transitioning existing applications to clouds or Multi-Clouds is not straightforward. A cloud is not merely a deployment environment to which existing software solutions can be transferred. It introduces novel characteristics not existing in traditional in-house deployment environments, such as a seemingly endless resource pool and the risk of unpredictable outages in external infrastructure [Varia 2011]. Hence, software applications need to be more scalable and fault tolerant so they can dynamically adapt to workload fluctuations by adequately allocating and releasing computing resources and addressing infrastructure failures autonomously and in a timely manner. Software engineers need to design for the cloud, not only deploy in the cloud. This is even more important when using multiple data centers situated in different legislative domains; constructed with different hardware, network, and software components; and prone to different environmental risks.

The key **contributions** of this work are (i) a design approach for interactive three-tier Multi-Cloud applications and (ii) adaptive dynamic provisioning and autonomous workload redirection algorithms ensuring that imperative constraints are met with minimal sacrifice in cost and QoE. We focus on the three-tier architectural pattern because it is pervasive and many enterprise systems follow it. Our approach does not modify the three-tier pattern itself, but rather introduces additional components to manage cross-cloud resource provisioning and workload distribution. This is essential because it allows the migration of existing applications to a Multi-Cloud environment. Also, new Multi-Cloud three-tier applications can be developed using a plethora of existing architectural frameworks thus leveraging proven technologies and existing know-how. The newly introduced components facilitate the implementation of three-tier systems that produce (i) increased availability and resilience to cloud infrastructure failure, (ii) legislation and regulation compliance, (iii) high QoE, and (iv) cost efficiency.

The rest of this article is organized as follows: In Section 2, we provide an overview of related works and compare them to ours. Section 3 details the targeted class of applications. Section 4 outlines our architecture. Section 5 motivates and details our algorithms for load balancing, autoscaling, and cloud selection. Our experimental settings and results are discussed in Section 6. In Section 7, we conclude the article and define pathways for future work.

2. RELATED WORK

Significant efforts have been made in the development of Multi-Cloud open-source libraries for different languages like JClouds [2014], Apache LibCloud [2014b], Apache DeltaCloud [2014a], SimpleCloud [2012], and Apache Nuvem [2014c]. All of them provide a unified API for the management of cloud resources (e.g., Virtual Machines [VMs] and storage), so that software engineers do not have to program against the specifics of each vendor's API. Although not providing application brokering (consisting of provisioning and scheduling) themselves, these libraries can be instrumental in the development of new cross-cloud brokering components. Similarly, services like RightScale [2014], Enstratius (formerly enStratus) [2014], Scalr [2014], and Kaavo [2014] only provide unified user interfaces, APIs, and tools for managing multiple clouds, and it is the clients' responsibility to implement appropriate provisioning and scheduling.

Apart from these Multi-Cloud libraries and services, the OPTIMIS [Ferrer et al. 2012], Contrail [Carlini et al. 2012], mOSAIC [Petcu et al. 2011], MODAClouds [Ardagna et al. 2012], and STRATOS [Pawluk et al. 2012] projects also facilitate Multi-Cloud application deployment. In all of these projects, the geographical locations of the serving data centers cannot be considered. Thus, often, it is not possible to implement legislation-aware application brokering. In contrast, in our approach, the *Entry Points* and *Data Center Control* layers enable legislation-compliant user routing to eligible clouds through a process called *matchmaking broadcast*. In addition, all of these projects only manage resource allocation and software component deployment, and none of them facilitates the distribution of the incoming workload to the allocated resources. Their components are only concerned with resource provisioning and set-up and do not deal with the load distribution and autoscaling of the application once it is installed. In contrast, in this work, we manage the incoming workload and dynamically provision resources accordingly through the components of the *Entry Points* and *Data Center Control* layers.

Furthermore, these projects are SLA-based, which means that the application brokering is specified in a Service Level Agreement (SLA) using a declarative formalism. The Cloud Standards Customer Council (CSCC) discusses in a technical report that SLAs currently offered by cloud providers are immature [2012]. Thus, to achieve flexible application brokering, these approaches rely on advances in the currently adopted SLA practices or the introduction of new brokering components that can interpret novel SLA formalisms. In contrast, our approach directly manages the underlying provisioning and mapping of workload to resources without relying on advances in SLA specifications, and thus it is applicable right away.

Cloud services like Route 53 [Amazon 2014c] and AWS Elastic Load Balancer (ELB) [Amazon 2014d] can distribute incoming users to servers in multiple data centers using standard load balancing techniques. AWS ELB can distribute workload among servers located in single or multiple AWS availability zones, but it cannot direct users to clouds of other providers. Route 53 is Amazon's Domain Name System (DNS) web service. It supports Latency Based Routing (LBR), which redirects incoming users to the AWS region with the lowest latency. Both Route 53 and ELB do not consider applications' regulatory requirements when selecting a data center site. Moreover, they do not consider the cost and degree of utilization of the employed resources within a data center. In contrast, our approach for directing users to cloud sites accounts for all these aspects.

3. PRELIMINARIES

By definition, an interactive three-tier application has three layers [Fowler 2003; Ramirez 2000; Aarsten et al. 1996]:

—**Presentation Layer:** Represents the user interface
—**Business/Domain Layer:** Features the main business logic; accesses and modifies the data layer
—**Data Layer:** Manages the persistent data

The presentation layer executes at the end user's site, not in the back-end servers, and thus we do not consider it. The domain layer consists of one or several Application Servers (AS). In Infrastructure as a Service (IaaS) cloud environments, they are hosted in separate VMs. The data layer consists of one or several database servers. In a Multi-Cloud environment, this software stack is replicated across all used cloud data centers. Clients arrive at one or several *entry points*, from where they are redirected to the appropriate data center to serve them.

The domain layer within a data center can scale horizontally by adding more AS VMs. For a given application, within a data center there is a load balancer that distributes the incoming requests to the AS servers. Every request arrives at the load balancer, which selects the AS to serve it. There are two types of three-tier applications in terms of the domain layer design: *stateful* and *stateless*. Stateful applications keep session data (e.g., shopping carts and user meta-data) in the memory of the assigned AS server. Hence, they require *sticky load balancing*, which ensures that all requests of a session are routed to the same server. Stateless applications do not keep any state/data in memory and therefore their requests can be routed to different AS servers.

The data layer often becomes the performance bottleneck because of requirements for transactional access and atomicity. This makes it hard to scale horizontally. As to the famous Consistency, Availability, and Partition Tolerance (CAP) theorem [Brewer 2000, 2012], a distributed architecture should balance between persistent storage consistency, availability, and partition tolerance. The field of distributed horizontally scaling databases has been well explored in recent years. For example Cattell [2010] surveyed more than 20 novel NoSQL and NewSQL distributed database projects. Traditional techniques like replication, caching, and sharding also allow for some level of horizontal scalability.

The eligible data caching and replication strategies are very much application specific, and it is impossible to incorporate them within a general framework encompassing all three-tier applications. In other words, the right balance among CAP requirements is domain inherent. For example, one application may require that data are not replicated across legislative regions, whereas another may allow it in order to achieve better availability. Therefore, in this work, we do not deal with application-specific data deployment. We investigate flexible provisioning and load distribution provided the data are already deployed with respect to the application-specific CAP requirements. It is the system architect's responsibility to design the data layer in a scalable way that obeys all domain-specific legislation rules so that it can be accessed quickly from the domain layer. This is a reasonable constraint because database design is usually the first step in a three-tier system design, and it often serves other applications (e.g., reporting and analytics) as well. Our approach ensures that once the data are deployed appropriately, users will be redirected accordingly and enough processing capacities will be present in the AS layer.

4. OVERALL ARCHITECTURE

4.1. Architectural Components

Figure 1 depicts the proposed architecture. We augment the traditional three-tier architectural pattern with two additional layers of components:

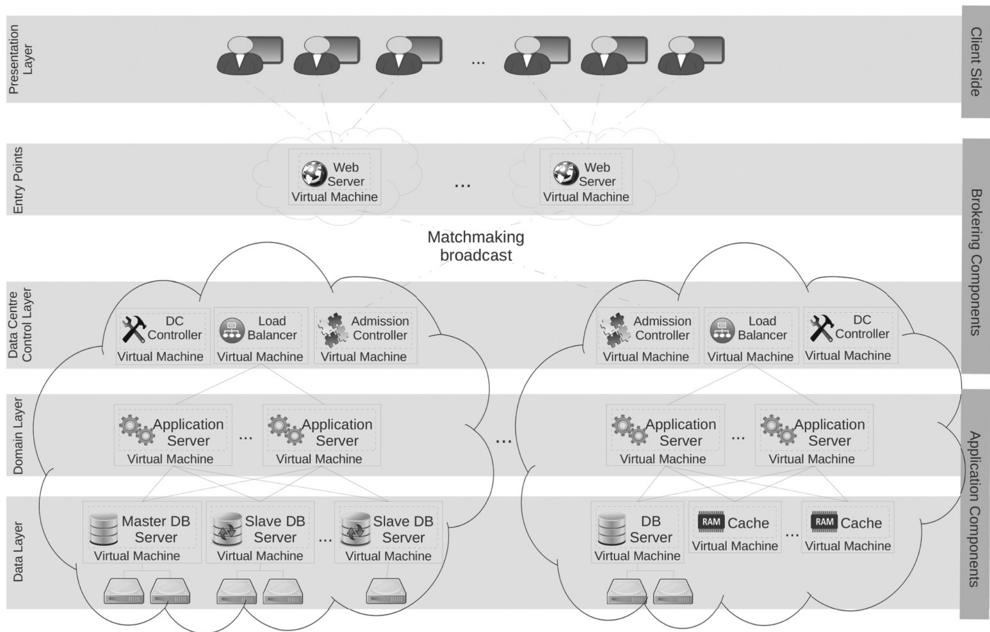

Fig. 1. Overall layered architecture. The brokering components manage the system's provisioning and workload distribution, whereas a standard three-tier software stack serves the end users.

—*Entry Point Layer:* Responsible for redirecting incoming users to an appropriate cloud data center to serve them

—*Data Center Control Layer:* Responsible for (i) providing information to the *entry point layer* regarding the suitability of a data center for a given user, (ii) monitoring and scaling of the provisioned resources within a data center, and (iii) directing the incoming requests

The *Entry Point Layer* consists of one or several VMs, which can be deployed in several data centers for better resilience. When users come to the system, they are initially served at an *entry point* VM. Based on the users' location, identity, and information about each data center, the *entry point* selects an appropriate cloud and redirects the user to it. After this, the user is served within the selected data center and has no further interaction with the *entry point*. We emphasize that the cross-cloud interactions between *entry points* and *admission controllers* happen only once, immediately after user arrival, and hence do not result in further communication delay as the user is being served.

At first glance, an *entry point* can be likened to a standard load balancer because it redirects users to serving data centers. However, standard load balancers redirect each user request, whereas *entry points* redirect users only upon arrival. Furthermore, *entry points* collaborate with the *admission controllers* to implement cloud selection respecting legislative, data location, cost, and QoE requirements, functions that are not implemented in standard load balancers.

In each data center, the *Data Center Control Layer* consists of three VMs:

—*Admission Controller:* Decides whether a user can be served within this data center and provides an estimation of the potential cost for serving him or her. Upon request,

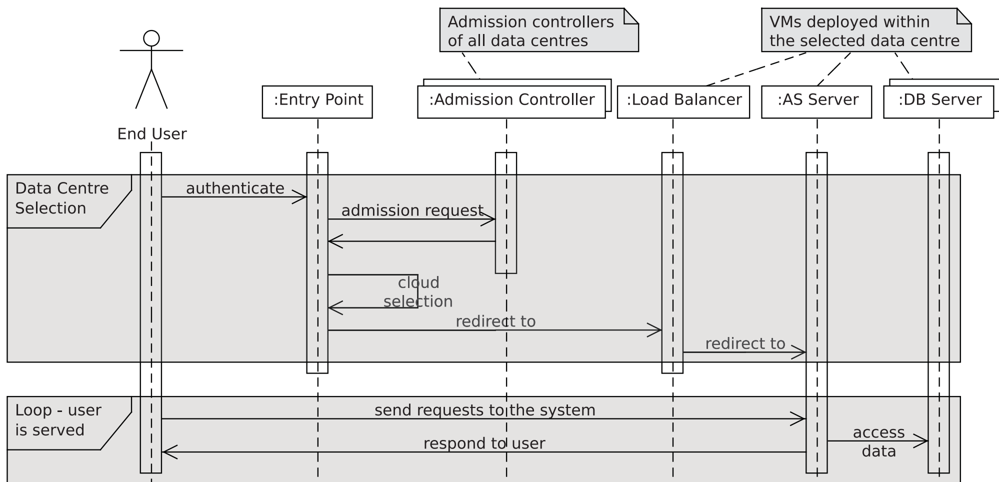

Fig. 2. Component interaction. Cloud site selection happens once, upon user arrival. Subsequent requests are handled within the selected data center.

it provides feedback to the *entry point* VMs to facilitate their choice of a data center for a user.

—*Load Balancer:* The standard *load balancer* component from the three-tier reference architecture. We consider it as a logical part of the *Data Center Control Layer* because it redirects requests to the application servers.

—*DC Controller:* Responsible for observing the performance utilization of the running AS servers and reactively shutting down or starting AS VM instances to meet resource demands at minimal cost.

In principle, the *DC Controller* and the *Load Balancer* VMs may be replaced by services like Amazon Auto Scaling [2014a] and AWS ELB [2014d]. Nevertheless, not all cloud providers have such services. Even if a provider offers autoscaling services, it is often not possible (unlike Amazon Auto Scaling) to monitor custom performance metrics (e.g., number of live application sessions). Moreover, in Section 5, we introduce novel algorithms for load balancing and autoscaling that, in conjunction, reduce cost and the probability of server overload. These are not implemented by current cloud providers and hence, for generality, we consider the usage of separate VMs for these purposes.

4.2. Component Interaction

Figure 2 depicts the interaction between components upon user arrival. In the first phase, the brokering components select an appropriate cloud site for the user based on his or her identity. As a first step, the user authenticates to one of the *entry points*. At this point, the *entry point* has the user's identity and geographical location (extracted from the IP address). As a second step, the *entry point* broadcasts the user's identifier to the *admission controllers* of all data centers. We call this step *matchmaking broadcast*.

There are no restrictions on the location of the *entry points*. Ideally, they should be positioned in a way that minimizes network latency effects during the *matchmaking broadcast*. One reasonable approach is to deploy each entry point in one of the used clouds, given that the clouds are already selected in a way that serves the expected user base with adequate latency.

Within each data center, the *admission controller* checks if the persistent data for this user are present. Additionally, each *admission controller* implements application-specific logic to determine which users can be served in the data center based on

regulatory requirements. The *admission controllers* respond to the *entry point* whether the user's data are present and if they are allowed (in terms of legislation and regulations) to serve the user. In the response, they also include information about costs within the data center.

Based on the *admission controllers'* responses, the *entry point* selects the data center to serve the user and redirects him or her to the *load balancer* deployed within it. The *entry point* filters all clouds that have the user's data and are eligible to serve him or her. If there is more than one such cloud, the *entry point* selects the most suitable with respect to network latency and pricing. If no cloud meets the data location and legislative requirements, the user is denied service.

After a data center is selected, the user is served by the AS and DB servers within the chosen cloud as prescribed by the standard three-tier architecture. He or she does not have any further interaction with the brokering components. Hence, we consider that AS servers deployed in a data center can only access DB servers in the same data center. This is a reasonable constraint because often there is no SLA concerning network speed and latency between data centers, and thus cross-cloud data access can lead to performance issues. Furthermore, transferring persistent data across the public Internet may be a breach of the legislative and policy requirements of many applications.

5. PROVISIONING AND WORKLOAD MANAGEMENT

5.1. Scalability within a Cloud

Currently, the practices for load balancing among a dynamic number of VMs in a cloud environment and among a fixed number of physical servers are the same: round robin or some of its adaptations. When using physical servers, one usually tries to distribute the load so that the servers are equally loaded, and all sessions are served equally well. In a cloud environment, if the number of AS VMs is insufficient, new ones can be provisioned dynamically. Similarly, if there are more than enough allocated AS VMs, some of them could be stopped to reduce costs. If the load of a stateful application is equally distributed among underutilized VMs, then no VM can be stopped without failing the sessions served there.

This is not an impediment for stateless applications because sessions are not bound to servers, and hence VMs can be stopped without causing service disruption. Thus, standard load balancing techniques like weighted round robin or "least connection" can be effective. However, in the case of stateful sessions, in order to stop an AS VM one needs to make sure it does not serve any sessions. One approach is to transfer all sessions from an AS VM to another one before shut down. However, this is not straightforward because active sessions together with their states need to be transferred without service interruption. A better approach is to balance the incoming workload in a way that consolidates the sessions in as few servers as possible without violating Quality of Service (QoS) requirements. This results in a maximum number of stoppable (i.e., not serving any sessions) servers. In essence, if the load balancer packs as many sessions as possible (without causing overload) onto a few servers, then the number of stoppable servers (not serving any sessions) will be maximal.

This idea is implemented in Algorithm 1. It defines a *sticky load balancing* policy; thus, after the first session's request is assigned to a server, all successive ones are assigned to it as well. It takes as input the newly arrived session s_i, the list of already deployed AS servers VM_{as}, and two ratio numbers in the interval $(0, 1)$—the CPU and RAM thresholds th_{cpu} and th_{ram}. As a first step in the algorithm, we sort the available AS VMs in a descending order with respect to their CPU utilization. Then we assign the incoming session to the first VM in the list whose CPU and RAM utilizations are below

ALGORITHM 1: Load Balancing Algorithm

input: s_i, th_{cpu}, th_{ram}, VM_{as}

```
sortDescendinglyByCPUUtilisation(VM_as);
hostVM ⟵ last element of VM_as;
for vm_i ∈ VM_as do
    vm_cpu ⟵ CPU utilisation of vm_i;
    vm_ram ⟵ RAM utilisation of vm_i;
    if vm_cpu < th_cpu and vm_ram < th_ram and !networkBuffersOverloaded() then
        hostVM ⟵ vm_i;
        break;
    end
end
assignSessionTo(s, hostVM)
```

th_{cpu} and th_{ram}, respectively, and whose input and output TCP network buffers/queues are not becoming overloaded. These buffer sizes are denoted by the Recv-Q and Send-Q values returned by the *netstat* command. To simplify the algorithm's definition, we have extracted this logic in a new boolean function *networkBuffersOverloaded*(). It simply checks if there is a TCP socket for which any of the ratios of the Recv-Q and Send-Q values to the maximum capacities of those queues is greater than 0.9. If there is no such server, the session is assigned to the least utilized one (line 2). An obvious postcondition of the algorithm is that a newly arrived session is assigned to the most utilized in terms of the CPU server whose CPU and RAM utilizations are under the thresholds. If there is no such server, the one with the least utilized CPU is used. By increasing the thresholds, we can achieve better consolidation of sessions at the expense of a higher risk of CPU or RAM contention, which may result in lower response time. On the contrary, when the thresholds are lower, the overall number of underutilized VMs will be higher. Therefore reasonable values for these thresholds are the autoscaling triggers (used by our autoscaling algorithm) that define if a server is overloaded.

The *DC controller* is responsible for adjusting the number of AS VMs accordingly. This implementation of the *load balancer* (Algorithm 1) allows the *DC controller* to stop AS VMs that serve no sessions. The *DC controller* is also responsible for instantiating new AS VMs when needed. Algorithm 2 details how this can be done when using on-demand VM instances. This algorithm is periodically executed every Δ seconds to ensure that the provisioned resources match the demand at all times. The input parameters of the algorithm are:

—t_{cur}—the current time of the algorithm call;
—tgr_{cpu}—CPU trigger ratio in the interval $(0, 1)$;
—tgr_{ram}—RAM trigger ratio in the interval $(0, 1)$;
—VM_{as}—list of currently deployed AS VMs;
—n—number of overprovisioned AS VMs to cope with sudden peaks in demand; and
—Δ—time period between algorithm repetitions.

If an AS VM's CPU or RAM utilization exceeds tgr_{cpu} and tgr_{ram} respectively, or some of its input/output TCP network buffers are becoming overloaded, we call this server overloaded. In the beginning of Algorithm 2 (lines 1–11) we inspect the statuses of all available AS VMs and note if they are overloaded or free (i.e., not serving any sessions).

In an online application, the resource demand can rise unexpectedly in the time periods between two subsequent executions of the scaling algorithm. Moreover, booting and setting up new AS VMs is not instantaneous and can take up to a few minutes

ALGORITHM 2: Scale Up/Down Algorithm

```
input: t_cur, tgr_cpu, tgr_ram, VM_as, n, Δ
nOverloaded ⟵ 0;
listFreeVms ⟵ empty list;
for vm ∈ VM_as ;                                   // Inspect the status of all AS VMs
do
    vm_cpu ⟵ CPU utilization of vm;
    vm_ram ⟵ RAM utilization of vm;
    if vm_cpu >= tgr_cpu or vm_ram >= tgr_ram or networkBuffersOverloaded() then
        nOverloaded ⟵ nOverloaded + 1;
    else if vm_i serves no sessions then
        listFreeVms.add(vm);
    end
end
nFree ⟵ length of listFreeVms;
nAS ⟵ length of VM_as;
allOverloaded ⟵ nOverloaded + nFree = nAS and nOverloaded > 0;
if nFree ≤ n ;                                     // Provision more VMs
then
    nVmsToStart ⟵ 0;
    if allOverloaded then
        nVmsToStart ⟵ n − nFree + 1;
    else
        nVmsToStart ⟵ n − nFree
    end
    launch nVmsToStart AS VMs
else
    nVmsToStop ⟵ 0 ;                               // Release VMs
    if allOverloaded then
        nVmsToStop ⟵ nFree − n;
    else
        nVmsToStop ⟵ nFree − n + 1
    end
    sortAscendinglyByBillingTime(listFreeVms);
    for i = 1 to nVmsToStop do
        billTime ⟵ billing time of listFreeVms[i];
        if billTime − t_cur < Δ then
            terminate listFreeVms[i];
        else
            break
        end
    end
end
```

depending on the underlying infrastructure. Hence, resources cannot be provisioned instantly in response to increased workload. If the workload spike is significant, this can result in server overload and performance degradation. One solution is to over-provision AS VMs, so that unexpected workload spikes can be handled. The n input parameter of the algorithm denotes exactly that—how many AS VMs should be over-provisioned to cope with unexpected demand.

As a postcondition of the algorithm execution, there should be at least $n+1$ free AS VMs if all other AS VMs are overloaded, or n otherwise. For example, in the special

case when $n = 0$, one AS VM is provisioned only if all others are overloaded. This is ensured by lines 16–24 of the algorithm.

Similarly, to avoid charges, some overprovisioned VMs should be stopped whenever their number exceeds n. However, it is not beneficial to terminate a running VM ahead of its next billing time. It is better to keep it running until its billing time in order to reuse it if resources are needed again. This is ensured by lines 25–40 of the algorithm. First, we sort the free VMs in ascending order with respect to their next billing time. Next, we iterate through the excessively allocated VMs and terminate only those for which the next billing time is earlier than the next algorithm execution time.

Last, in the previous discussion, we assumed that the application is stateful (i.e., it maintains contextual user information in the memory of the AS server). For scalability reasons, many applications are stateless, or they store session state in an external in-memory cache like Amazon ElastiCache [2014b]. Algorithm 2 can handle this type of applications as well by considering each AS server to be assigned 0 sessions at all times. This is reflective of the main characteristic of stateless applications, in that each request can be served on a different server because no session state is kept in the servers' memory. Consequently, in the algorithm, all AS servers that are not overloaded will be considered free (lines 9–11) and will be viable for termination. Therefore, our approach encompasses both stateful and stateless applications.

5.2. Data Center Selection

We can largely classify the requirements for data center selection as constraints and objectives. Constraints should not be violated under any circumstances. In this work, we consider the following constraints: (i) users should be served in data centers that are compliant with regulatory requirements, and (ii) users should be served in data centers containing their data in the application's data layer. The system should prefer to deny service than to violate these constraints. In contrast, the system can continue to serve a user even if an objective is not optimized. We consider the following objectives: (i) cost minimization and (ii) latency minimization. In other words, upon a user's arrival, the *entry point* extracts those data centers that satisfy the constraints and selects the most suitable among them in terms of latency and cost.

Although cost minimization is a natural goal of cloud clients, it should not be pursued at the expense of end user QoE, and hence we must balance between the two objectives. The maximum acceptable latency between users and data centers can be considered as a part of the application's SLA. Thus, we can choose the optimal data center, in terms of cost, whose network latency is less than the one predefined in the SLA.

Algorithm 3 implements this idea and details the data center selection procedure. The algorithm selects clouds for multiple users at once. Hence, users arriving at the system at approximately the same time can be dispatched to serving clouds in batch thus avoiding excessive cross-cloud communication. The input parameters of the algorithm are:

—*users*: Identifiers of the users for which the *entry point* should select a cloud
—*timeout*: Period after which, if a data center's *admission controller* has not responded, it is discarded
—*clouds*: A list of the used data centers. For each of them, we can obtain the IP addresses of the *admission controller* and the *load balancer*.
—$latency_{SLA}$: SLA for the network latency between a user and the serving data center

In the beginning of the algorithm (lines 1–4), the *entry point* asynchronously broadcasts all users' identifiers to the clouds' *admission controllers*. After that, the *entry point* waits until all contacted *admission controllers* respond or the timeout period

ALGORITHM 3: Cloud Site Selection Algorithm

```
input: users, timeout, clouds, latency_SLA
// Broadcast users' data to admission controllers
for c_i ∈ clouds do
    ac_i ⟵ IP address of c_i's admission controller;
    send to ac_i users' identifier;
end
wait timeout seconds or until all clouds respond;
for u_i ∈ users do
    clouds_accept ⟵ clouds eligible to serve u_i;
    sortAscendinglyByPrice(clouds_accept);
    selectedCloud ⟵ null;
    selectedLatency ⟵ +∞ ;
    for c_i ∈ clouds_accept do
        latency ⟵ latency between u and c_i;
        if latency < latency_SLA then
            selectedCloud ⟵ c_i;
            break;
        else if selectedLatency > latency then
            selectedCloud ⟵ c_i;
            selectedLatency ⟵ latency;
        end
    end
    if selectedCloud = null then
        Deny Service;
    else
        lb ⟵ IP of load balancer in selectedCloud;
        redirect u to lb;
    end
end
```

elapses (line 5). At this stage, unresponsive clouds whose *admissions controllers* fail to respond within the timeout are discarded.

For each user, the response of the clouds' *admission controllers* includes (i) a boolean value, whether the cloud is eligible to serve the user and (ii) an estimation of the cost for serving a user. Based on this input, for every user, the *entry point* retains the clouds eligible to serve him or her (line 7). If no eligible cloud is present, the user is denied service. Otherwise, the cloud that has the smallest cost and provides latency below the SLA requirement is selected (lines 9–20). If there is no eligible cloud meeting the network latency SLA, the one with the lowest network latency to the user is selected (lines 16–19).

Note that the decision of whether a cloud site is eligible for a given user is application specific. For some applications with no additional privacy, security, and legislative requirements, all clouds may be eligible for all users. In others, certain users will have to be served within a specific legislative domain or within certified data centers based on their nationality. We assume that this application-specific eligibility logic is implemented in *admission controllers* by application developers.

Algorithm 3 uses the network latency between the end user and the prospective serving data centers. Hence, the *entry point* needs to evaluate the latencies between them based on their IP addresses. By latency, we denote only the network latency. We do not try to estimate the entire response time consisting of network and server

delays. We argue that this simplification does not reduce the generality of our approach because for an interactive three-tier application, the server delays should be small and similar in all clouds provided there is no significant resource contention. In this case, the variable part of the overall delay is the network latency. In Algorithm 2, we make sure the domain layer scales horizontally and that enough resources are present at all times. If contention does occur within a cloud site because of either a nonscalable DB layer or an inappropriate choice of parameters for Algorithm 2, then we provide a back-off mechanism through the cost estimation, as discussed later. Hence, we minimize the probability of resource contention within the Multi-Cloud setup, and therefore servers' delays should be small and similar in all cloud sites.

An approximation of the network latency can be achieved in two steps. First, we can identify the geographical locations (longitude and latitude) of the user and a given cloud based on his or her IP address. For this, we use the GeoLite [2014] database, mapping IP addresses to geospatial coordinates. As a second step, we compute the latency between a user and a cloud based on the extracted coordinates by using the PingER [2014] service. PingER is an end-to-end Internet Performance Measurement (IEPM) project that constantly records the network metrics among more than 300 hosts positioned worldwide. The geospatial coordinates of each host are provided. To approximate the latency between a user and a cloud, we select the three pairs of PingER hosts that are closest to the user and the cloud, respectively, and define the latency as a weighted sum of the three latencies between the hosts in these three pairs. The weights are defined proportionally to the proximity of the hosts to the user and the cloud. To compute the distance between the geospatial positions, we use the well- known Vincenty's formulae. The data from both GeoLite and PingER can be downloaded and used offline. If latest up-to-date Internet performance data are needed, they can be periodically downloaded and updated automatically.

The last missing piece of information is the cost evaluation for serving a user by a cloud (used in line 8), performed by the *admission controllers*. The difficulty here is to define a unified cost evaluation for different clouds with different pricing policies and different VM types and performance.

First, if the application's infrastructure within a data center is overloaded and it should not accept further users, it returns $+\infty$ as a cost estimation. One reason for an overload may be a lack of scalability in the DB layer. As discussed, it can be hard to scale horizontally, and given significant workload, the center can be easily overloaded. Another reason may be a bottleneck in the data center infrastructure; for example, internal network congestion may threaten to slow down the application's inter-tier communication. This could be easily detected in the case of a private data center. In a public data center, obtaining such information may be more difficult because the cloud provider would have to expose such internal performance data to its clients. By returning $+\infty$ cost to the *entry point,* the *admission controller* ensures that users are sent to this cloud only as a last resort. Hence, *admission controllers* use the cost as a back-off mechanism.

As a first step in session cost estimation, we define $p(vm_i)$ to be the price per minute of a virtual machine vm_i. For cloud providers that charge for longer intervals (e.g., an hour, as in Amazon AWS), we compute this value by dividing by the number of minutes in a charge period. For each virtual machine vm_i, based on its current utilization and the number of currently served sessions, we can approximate how many sessions $f(vm_i)$ it will be able to serve if fully utilized:

$$f(vm_i) = \frac{numSessions(vm_i)}{max(util_{cpu}(vm_i), util_{ram}(vm_i))} \quad . \tag{1}$$

Therefore the term $p(vm_i)/f(vm_i)$ is representative of the session cost per minute in vm_i. To achieve better estimation, we average the cost estimations of all AS servers V that currently serve sessions in the cloud. If no sessions are served in the cloud, we use the last successful estimation for this data center or 0 if there has not been such. Equation (2) summarizes the previous discussion:

$$\textit{session cost per minute} = \begin{cases} +\infty & \text{if } \textit{overloaded} \\ \textit{previous estimation} & \text{if } V = \emptyset \\ \frac{\sum_{vm_i \in V} p(vm_i)/f(vm_i)}{|V|} & \text{otherwise.} \end{cases} \quad (2)$$

In the preceding discussion, for each VM vm_i we used only the price per minute ($p(vm_i)$), number of sessions, and its CPU and memory utilizations. Therefore our cost evaluation can be used even if the types of the VMs are different as long as we can evaluate these characteristics. It is also worthwhile noting that this cost estimation strategy is a heuristic forecast of the future cost incurred by a user because we cannot know in advance how long the user will use the system, what exactly will be his or her actions, and the like.

5.3. Fault Tolerance

Within the given architecture, a data center outage can be seamlessly overcome by incorporating time-outs in the *entry points*. If an *admission controller* does not make a timely reply to the *matchmaking broadcast*, the *entry point* does not consider its respective cloud.

Within a cloud, the *DC controller* manages how the AS VMs are instantiated and stopped in order to meet QoS requirements with minimal costs. Doing so, it also monitors the AS VMs in the data center and restarts them upon failure. In this setting, it is obvious that a failure of the *DC Controller* would disable the fault tolerance and scalability of the architecture. Hence, the *admission controller* and the *load balancer* run background threads that check the status of the *DC controller* and restart it upon failure.

VMs can take up to a few minutes to boot. A failure of the *load balancer* and the *admission controller* would mean that no users can be served in this data center during such an outage. Thus, applications requiring high availability can have multiple *load balancers* and *admission controllers* working in parallel to achieve resilience against such failure.

6. PERFORMANCE EVALUATION

Our approach to application provisioning and workload distribution is generic, and testing it with all possible middleware technologies, workloads, cloud offerings, and data center locations is an extremely laborious task. In this section, we demonstrate how, under typical workload and set-up, our approach meets imperative requirements like legislation compliance with only minimal losses in terms of latency and cost.

To validate our work, we use the CloudSim discrete event simulator [Calheiros et al. 2011], which has been used in both industry and academia for performance evaluation of cloud environments and applications. We use one of the latest CloudSim extensions that allows modeling, simulation, and performance evaluation of three-tier applications in Multi-Cloud environments [Grozev and Buyya 2013].

6.1. Experiment Setting

In our experimental setup, we create four cloud data centers. We model the first two with the characteristics of Amazon EC2. We position one of them in Dublin, Ireland and

Table I. Simulation Parameters

Parameter	Value	Component/Algorithm
th_{CPU}	0.7	Load balancer (AS Server Selection)
th_{RAM}	0.7	Load balancer (AS Server Selection)
tgr_{CPU}	0.7	DC Controller (Autoscaling)
tgr_{RAM}	0.7	DC Controller (Autoscaling)
n	1	DC Controller (Autoscaling)
Δ	10 sec	DC Controller (Autoscaling)
$latency_{SLA}$	30 ms	Entry point (Cloud Selection)

the other one in New York City. These are actual locations of EC2 availability zones. Later, we call these data centers *DC-EU-E* and *DC-US-E*, respectively. We assign the VMs from these data centers IP address from Dublin and New York, respectively, which are extracted from GeoLite. All VMs we allocate in these data centers have the performance characteristics and the price of EC2 *m1.small* instances with Linux in the respective AWS regions. We model the VM start-up times based on the empirical performance study by Mao and Humphrey [2012]. Just as in Amazon EC2, the on-demand VM billing in *DC-EU-E* and *DC-US-E* is done per hour.

To demonstrate the usage of heterogeneous cloud resources from multiple cloud providers, we model the other two data centers after Google Compute Engine. We position them in Hamina, Finland and Dalles, Oregon because these are actual locations of Google data centers, and we assign all their VMs IP addresses from these locations. We call these data centers *DC-EU-G* and *DC-US-G*. All VM characteristics and prices are modeled after the *n1-standard-1-d* VM type in the respective locations. Because Google Compute Engine is a new cloud offering, there is no statistical analysis of its VM booting times. Thus, in our simulation, we consider the start-up time of an *n1-standard-1-d* VM to be the same as the one of an EC2 *m1.small* VM. As in Google Compute Engine, VMs in *DC-EU-G* and *DC-US-G* are billed in 1-minute increments, and all VMs are charged for 10 minutes at least.

In our simulation, we deploy the aforementioned three-tier architecture and brokering components as described in the previous sections. We model one *entry point* VM in each data center. To demonstrate how our approach handles resource contention in the data layer, in the experiments, we assume that in each cloud the DB layer is static (i.e., not scalable) and consists of two DB servers holding equally sized database shards. Table I summarizes the values of the algorithms' parameters that we use in the simulation.

6.2. Experimental Application and Workload

We base our experimental workload on the Rice University Bidding System (RUBiS) benchmarking environment [RUBiS 2014; Amza et al. 2002]. RUBiS implements an e-commerce dynamic web site similar to eBay.com and follows the three-tier architectural pattern by having AS and DB servers.

RUBiS's main and (to the best of our knowledge) newest competitor in the area of three-tier application benchmarking is CloudSuite's CloudStone [2014]. Unlike RUBiS, which has a simple synchronous web interface lacking any JavaScript, CloudStone implements a more sophisticated asynchronous (i.e., AJAX) web user interface. Although this is important for evaluating how end users interact with a system, in this work, we are interested in how to provision for and load balance the incoming server-side requests. RUBiS follows the guidelines of the TPC-W [2002] specification of the Transaction Processing Performance Council (TPC), which is an industry standard for testing three-tier e-commerce systems. The RUBiS client implements features like end user "think times" and page transitions in accordance with the predefined statistical

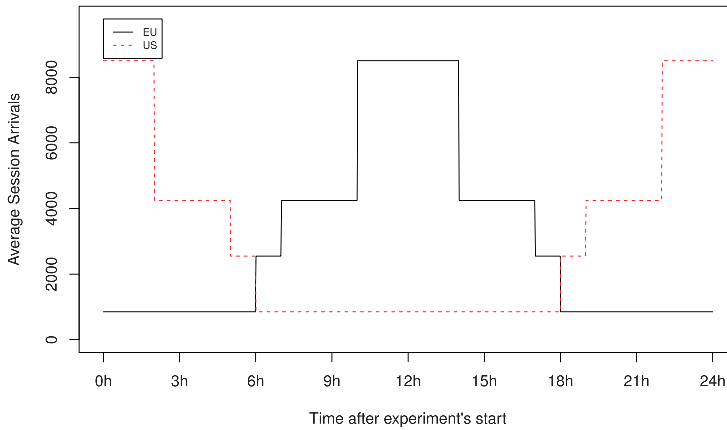

Fig. 3. User arrival frequencies per minute in the entry points of the EU data centers (*DC-EU-E*, *DC-EU-G*) and the US ones (*DC-US-E*, *DC-US-G*).

distributions defined in TPC-W. Thus, it is our method of choice for a typical three-tier application in terms of incoming server request patterns per user compared to CloudStone, which does not follow a public specification.

The RUBiS workload consists of sessions, each of which consists of a string of user requests. We have deployed in a local virtualized environment the PHP version of RUBiS with a standard nonclustered MySql database in the backend, and we run a test with 100 concurrent sessions. During the test, we monitor how the performance utilizations (in terms of CPU, RAM, and disk) of the servers change over time as a result of the executed workload. Based on that, we define the performance utilizations over time of a typical RUBiS session. We will not describe the exact procedure for extracting a session performance model because our previous work [Grozev and Buyya 2013] details this procedure and demonstrates experimentally the validity of the extracted model. We use this derived session performance model for our simulation.

The number of incoming sessions over a short time period can be well modeled with Poisson distribution with a constant mean λ [Cao et al. 2003; Robertson et al. 2003]. However, over longer time periods, the frequencies of user arrivals can change and are rarely constant. Hence, the number of session arrivals over time can be represented as a Poisson distribution over a frequency function of time $\lambda(t)$, which represents the variations in session arrival frequencies [Grozev and Buyya 2013].

Our experiment has a duration of 24 hours with a workload that is more intensive during working hours and lower otherwise. We model the user arrival frequencies in the *entry points* of the two European (EU) data centers to be the same. The arrival frequencies in the US data centers are the same as those in the EU ones, only “shifted” by 12 hours to represent the time zone difference. Figure 3 depicts how the arrival frequencies per minute (i.e., $\lambda(t)$) in the entry points of the European and the US data centers change over time.

In our simulation, each user/session is assigned an IP address that, as explained previously, can be used to approximate the user’s physical location and the latencies to the candidate data centers. The GeoLite [2014] database provides IP ranges for every country. In the simulation, whenever we model the arrival of a user in a US data center, we take a random US IP from GeoLite. Similarly, all users arriving in the EU data centers are assigned random IP addresses from EU countries.

To demonstrate how our system handles regulatory requirements, we introduce an additional legislative constraint. In the simulation, we assign a citizenship to each user

and impose the requirement that a user with US citizenship should be served in a US data center and an EU citizen should be served in the EU. As discussed, we implement this logic in the *admission controllers*. We assign US citizenship to 10% of the users arriving in the EU entry points and EU citizenship to 10% of the users arriving in the US clouds. Furthermore, in our simulation, the data of all EU citizens are replicated in both EU data centers, and the data of all US citizens are replicated in both US cloud sites. Therefore, an EU or a US citizen can be served in any EU or US data center, respectively.

6.3. Baseline Approach

We compare our approach to a baseline method that uses standard industry practices. More specifically, we have implemented a baseline simulation that distributes incoming users to those data centers that can serve them with the lowest latency, similarly to the Route 53 LBR service [Amazon 2014c]. Following the design of the AWS ELB [Amazon 2014d], within each data center we implement sticky load balancing that assigns new sessions to running AS servers following the round-robin algorithm. Last, in our baseline simulation, we implement automatic autoscaling following the design of AWS AustoSale [Amazon 2014a]. More specifically, if all AS servers within a data center reach a CPU utilization of more than 80%, a new AS server is started. If an AS server reaches a CPU utilization of below 10%, it is stopped. We have also implemented a *cool-down* period of 2.5 minutes. Just as in AWS AustoSale, we do not allow for two consequent autoscaling actions to happen within a period shorter than the *cool-down* period.

6.4. Results

Figure 4 depicts the number of served sessions in each data center over time. In the diagram, a session is classified as *failed* if some of the servers handling it failed (e.g., due to out-of-memory errors). A session is *rejected* if it is assigned to a data center that is not eligible to serve it. In our simulation, this happens if a US citizen is assigned to a European data center or vice versa. Otherwise, a session is considered *served*.

From Figure 4, we can see that the baseline approach redirects many fewer sessions to data centers *DC-EU-G* and *DC-US-G* in comparison to the others. This is because of the location of these data centers and the end users. As described, in our experiment, all IP addresses within the EU and US are likely to be used as sources of sessions with the same probability. As to the GeoLite database and the PingER service, there are many more IP addresses located nearby and with lower latency to Dublin, Ireland and New York City than to Hamina, Finland and Dalles, Oregon. Therefore, the baseline approach redirects the majority of incoming sessions to these data centers. Because the data layer cannot scale up, this leads to resource contention during peak workload periods (10h–14h in the EU data centers and 22h–24h, 0h–2h in the US). This in turn causes congestion in the DB servers resulting in a slowdown in session serving. As a result, the number of concurrently served sessions is increased significantly, causing AS servers keeping in memory the sessions' states to fail with "out-of-memory" errors (in the case of *DC-US-E*) or to degrade response time (in the case of *DC-EU-E*).

Another reason for session failure in the baseline approach is autoscaling, which terminates AS servers with low utilization even if they serve sessions. This is visible in the case of *DC-EU-E* during the 16h–24h period and in *DC-US-E* during the 0h–4h period, when the scaling down causes several session failures because sessions are stateful. Our approach terminates servers only if they do not serve any sessions and thus reduces the number of session failures for stateful applications. Consequently, the overall rate of session failures in the baseline is approximately 7%.

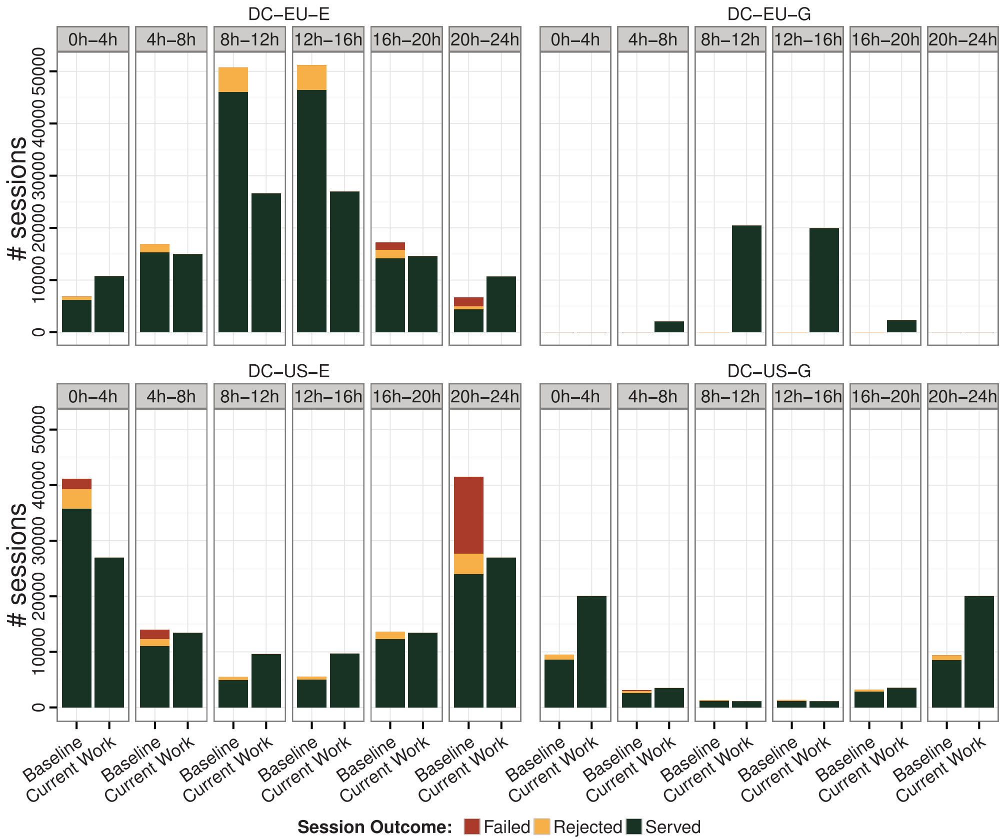

Fig. 4. Session outcome over time.

In contrast to the baseline approach, during the workload peak periods, our approach redirects many sessions to *DC-EU-G* and *DC-US-G* even though they may not be optimal in terms of cost or latency. This is because the cost of a data center is evaluated as $+\infty$ if it is overloaded (see Equation (2)), and this is used by the cloud selection Algorithm 3. As a result, our approach minimizes session failure by diverting users from overloaded data centers to alternative ones. During off-peak hours, our approach also redirects most users to *DC-EU-E* and *DC-US-E*, which, as discussed, is optimal in terms of latency.

Furthermore, the baseline approach does not consider the stated regulatory requirements during the cloud selection stage and only uses latency as selection criteria. Thus, about 10% of the incoming sessions are redirected to ineligible clouds and are rejected. In contrast, our approach takes this into consideration and redirects users to eligible data centers, even if this means suboptimality in terms of latency and cost.

A session delay is defined as the sum of the latency delay and the execution delay. The latency delay is the time lost in network transfer between a user and the cloud during a session. Execution delay is the time lost due to resource contention (e.g., CPU preemption) on the server side. The session delay is a measurement of the end user experience. The simulation environment allows us to measure execution delay [Grozev and Buyya 2013], and we can compute the latency delay based on the latency and the average number of interactions/requests during a session.

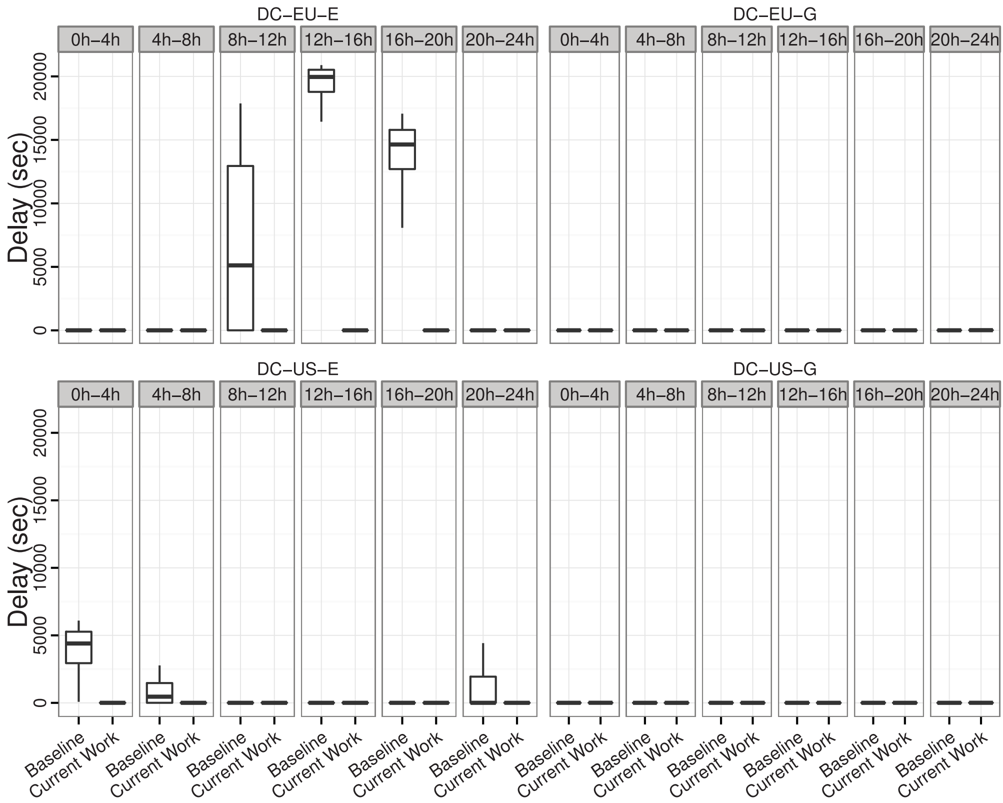

Fig. 5. Session delays in seconds.

Figure 5 depicts the session delays of those users served in the European and US data centers. The delays in *DC-EU-E* and *DC-US-E* during the peak workload periods of the baseline approach significantly exceed those of our approach. Similar to the session failures, this is caused by resource contention in the DB layer. In contrast, all delays in *DC-EU-G* and *DC-US-G* are insignificant since the baseline approach redirects very few users there and therefore resource contention is small. The delays in *DC-US-E* are smaller than those in *DC-EU-E* because the failures there were much more frequent, and therefore fewer sessions were actually measured (see Figure 4). In our approach, DB layer contentions are mitigated because users are redirected to alternative data centers whenever the data layer in a given cloud is overloaded. Moreover, the overall session delays in our approach are less than 10s at all times in all data centers, showing that the effect of selecting a cloud with less than optimal latency in some cases is small.

Figure 6 shows the distributions of the achieved latencies between clients and clouds using the baseline and our approaches. The average baseline latency is lower by approximately 10ms than the one in our approach because the baseline method greedily selects the data center with lowest latency. Also, our approach honors the stated regulatory requirements, and hence 10% of the users are served overseas, which contributes to the increased average latency. Still, in our approach, the mean, median, and interquartile range of the latencies is below the stated SLA of 30ms thus providing for adequate QoE. End users experience the network latency through application response delays. In our approach, the average network delay is higher, but the execution delay is much lower, resulting in lower overall delay (see Figure 5) and better QoE.

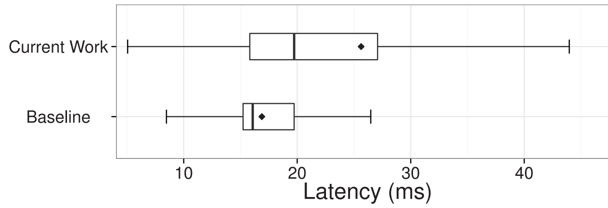

Fig. 6. Network latency between users and serving data centers in milliseconds. The mean is denoted with a rhombus.

Last, the overall cost incurred by our approach is about 28% more than the cost of the baseline. First, this can be attributed to the number of rejected and failed (approximately 17%) sessions in the baseline experiment. Since the baseline approach effectively served many fewer sessions, it needed fewer servers and therefore its cost is lower. Also, in order to prevent failure, our approach redirects many sessions to more expensive data centers thus resulting in increased overall cost.

7. CONCLUSION AND FUTURE WORK

In this work, we introduced a novel approach for adaptive resource provisioning and workload distribution of three-tier applications across clouds. It encompasses all aspects of resource management and workload redirection including (i) cloud selection, (ii) load balancing, and (iii) autoscaling. We introduced new architectural components and algorithms that ensure that imperative requirements like regulation compliance and high availability are not violated without sacrificing too much cost and end-user QoE. To validate our approach, we performed simulations with realistic cloud data center settings, VM types, costs, and network characteristics derived from a real-life benchmarking applications, cloud providers, and Internet monitoring services. We compared our approach to a baseline approach that follows current industrial best practices. Results show that our approach is a significant improvement over the baseline in terms of achieved availability, accumulative session delay, and regulatory compliance while maintaining acceptably low cost and latency between users and serving data centers.

In the future, we plan to extend our algorithms to utilize a mixture of reserved and on-demand VM instances. Also, we intend to investigate how to automatically select the most appropriate (in terms of performance and cost) type of VM in each cloud. Another interesting extension would be to consider data center availability (defined in the provider's SLAs or by a third party) in the cloud selection Algorithm 3.

ACKNOWLEDGMENTS

We thank Rodrigo Calheiros, Amir Vahid Dastjerdi, Adel Nadjaran Toosi, Atefeh Khosravi, Yaser Mansouri, Chenhao Qu, and Deborah Magalhães for their comments on improving this work. We also thank Amazon.com, Inc. for their support through the AWS in Education Research Grant.

REFERENCES

A. Aarsten, D. Brugali, and G. Menga. 1996. Patterns for three-tier client/server applications. In *Proceedings of Pattern Languages of Programs (PLoP'96)*.

Amazon. 2014a. Amazon Auto Scaling. (Feb. 3 2014). http://aws.amazon.com/autoscaling/.

Amazon. 2014b. Amazon ElastiCache. (Feb. 3 2014). http://aws.amazon.com/elasticache/.

Amazon. 2014c. Amazon Route 53. Retrieved from http://aws.amazon.com/route53/.

Amazon. 2014d. Elastic Load Balancing. Retrieved from http://aws.amazon.com/elasticloadbalancing/.

Amazon. 2014e. Summary of the Amazon EC2 and Amazon RDS Service Disruption. Retrieved from http://aws.amazon.com/message/65648/.

Amazon. 2014f. Summary of the AWS Service Event in the US East Region. Retrieved from http://aws.amazon.com/message/67457/.

C. Amza, A. Chanda, A. L. Cox, S. Elnikety, R. Gil, K. Rajamani, W. Zwaenepoel, E. Cecchet, and J. Marguerite. 2002. Specification and implementation of dynamic Web site benchmarks. In *Proceedings of the IEEE International Workshop on Workload Characterization*. 3–13.

Apache Foundation. 2014a. Apache Delta Cloud. Retrieved from http://deltacloud.apache.org/.

Apache Foundation. 2014b. Apache Libcloud. Retrieved from http://libcloud.apache.org/.

Apache Foundation. 2014c. Apache Nuvem. Retrieved from http://incubator.apache.org/nuvem/.

D. Ardagna, E. Di Nitto, P. Mohagheghi, S. Mosser, C. Ballagny, F. D'Andria, G. Casale, P. Matthews, C.-S. Nechifor, D. Petcu, A. Gericke, and C. Sheridan. 2012. MODAClouds: A model-driven approach for the design and execution of applications on multiple Clouds. In *Proceedings of the Workshop on Modeling in Software Engineering (MISE'12)*. 50–56. DOI:http://dx.doi.org/10.1109/MISE.2012.6226014

M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, and R. H. Katz. 2009. *Above the Clouds: A Berkeley View of Cloud Computing*. Technical Report. Electrical Engineering and Computer Sciences, University of California at Berkeley.

J. A. Bowen. 2011. Legal issues in cloud computing. In *Cloud Computing: Principles and Paradigms*, R. Buyya, J. Broberg, and A. M. Goscinski (Eds.). Wiley Press, Chapter 24, 593–613.

E. Brewer. 2000. Towards robust distributed systems. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, Vol. 19. ACM, New York, NY, US, 7–10.

E. Brewer. 2012. CAP twelve years later: How the "Rules" have changed. *Computer* 45, 2 (2012), 23.

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

R. Calheiros, R. Ranjan, A. Beloglazov, C. De Rose, and R. Buyya. 2011. CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience* 41, 1 (January 2011), 23–50.

J. Cao, M. Andersson, C. Nyberg, and M. Kihl. 2003. Web server performance modeling using an M/G/1/K*PS queue. In *Proceedings of the 10th International Conference on Telecommunications (ICT'03)*, Vol. 2. 1501–1506.

E. Carlini, M. Coppola, P. Dazzi, L. Ricci, and G. Righetti. 2012. Cloud federations in contrail. In *Proceedings of Euro-Par 2011: Parallel Processing Workshops*, Michael Alexander et al. (Eds.). Lecture Notes in Computer Science, Vol. 7155. Springer, Berlin, 159–168.

R. Cattell. 2010. Scalable SQL and NoSQL data stores. *SIGMOD Record* 39, 4 (May 2010), 12–27.

CloudSuite. 2014. CloudSuite's CloudStone. Retrieved from http://parsa.epfl.ch/cloudsuite/web.html.

CSCC Workgroup. 2012. *Practical Guide to Cloud Service Level Agreements Version 1.0*. Technical Report. Cloud Standards Customer Council (CSCC).

Ebay. 2014. Ebay. (Feb. 3 2014). http://www.ebay.com/.

Enstratius. 2014. Enstratius. Retrieved from https://www.enstratius.com/.

A. J. Ferrer, F. Hernández, J. Tordsson, E. Elmroth, A. Ali-Eldin, C. Zsigri, R. Sirvent, J. Guitart, R. Badia, K. Djemame, W. Ziegler, T. Dimitrakos, S. Nair, G. Kousiouris, K. Konstanteli, T. Varvarigou, B. Hudzia, A. Kipp, S. Wesner, M. Corrales, N. Forgó, T. Sharif, and C. Sheridan. 2012. OPTIMIS: A holistic approach to cloud service provisioning. *Future Generation Computer Systems* 28, 1 (2012), 66–77.

M. Fowler. 2003. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional.

GeoLite. 2014. GeoLite2 Free Downloadable Databases. Retrieved from http://dev.maxmind.com/geoip/legacy/geolite/.

Google. 2014. Post-mortem for February 24th, 2010 Outage. Retrieved from https://groups.google.com/group/google-appengine/browse_thread/thread/a7640a2743922dcf?pli=1.

N. Grozev and R. Buyya. 2012. Inter-Cloud architectures and application brokering: Taxonomy and survey. *Software: Practice and Experience* 44, 3 (2012), 369–390.

N. Grozev and R. Buyya. 2013. Performance modelling and simulation of three-tier applications in cloud and multi-cloud environments. *Computer Journal* (in press) (2013).

IBM. 2013. *IBM Takes Australian Open Data onto Private Cloud*. Technical Report. IBM.

JClouds. 2014. JClouds. Retrieved from http://www.jclouds.org/.

Kaavo. 2014. Kaavo. Retrieved from http://www.kaavo.com/.

M. Mao and M. Humphrey. 2012. A performance study on the VM startup time in the cloud. In *Proceedings of the 5th IEEE Conference on Cloud Computing (CLOUD'12)*. 423–430.

P. Mell and T. Grance. 2011. *The NIST Definition of Cloud Computing*. Special Publication 800-145. National Institute of Standards and Technology.

Microsoft. 2014. Windows Azure Service Disruption Update. Retrieved from http://blogs.msdn.com/b/windowsazure/archive/2012/03/01/windows-azure-service-disruption-update.aspx.

P. Pawluk, B. Simmons, M. Smit, M. Litoiu, and S. Mankovski. 2012. Introducing STRATOS: A cloud broker service. In *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD'12)*. IEEE.

D. Petcu. 2013. Multi-Cloud: Expectations and current approaches. In *Proceedings of the International Workshop on Multi-Cloud Applications and Federated Clouds (Multi-Cloud'13)*. ACM, New York, NY, 1–6.

D. Petcu, C. Crăciun, M. Neagul, S. Panica, B. Di Martino, S. Venticinque, M. Rak, and R. Aversa. 2011. Architecturing a sky computing platform. In *Proceedings of the International Conference towards a Service-Based Internet (ServiceWave'10)*, Michel Cezon and Yaron Wolfsthal (Eds.). Lecture Notes in Computer Science, Vol. 6569. Springer-Verlag, Berlin, 1–13.

PingER. 2014. Ping End-to-End Reporting. Retrieved from http://www-iepm.slac.stanford.edu/pinger/.

A. Ramirez. 2000. Three-tier architecture. *Linux Journal* 2000, 75, Article 7.

RightScale. 2014. RightScale. Retrieved from http://www.rightscale.com/.

A. Robertson, B. Wittenmark, and M. Kihl. 2003. Analysis and design of admission control in Web-server systems. In *Proceedings of the American Control Conference*, Vol. 1. 254–259.

RUBiS. 2014. RUBiS: Rice University Bidding System. Retrieved from http://rubis.ow2.org/.

Scalr. 2014. Scalr. Retrieved from http://scalr.net/.

Simple Cloud. 2012. Simple Cloud API. Retrieved from http://simplecloud.org/.

Transaction Processing Performance Council (TPC). 2002. *TPC BENCHMARK W (Web Commerce)*. Specification, version 1.8. Transaction Processing Performance Council (TPC).

J. Varia. 2011. Best practices in architecting cloud applications in the AWS cloud. In *Cloud Computing: Principles and Paradigms*, R. Buyya, J. Broberg, and A. M. Goscinski (Eds.). Wiley, 459–490.

Received February 2014; revised April 2014; accepted June 2014