Noname manuscript No.
(will be inserted by the editor)

Regulations and Latency Aware Load Distribution of Web Applications in Multi-Clouds

Nikolay Grozev · Rajkumar Buyya

Received: October 2015 / Accepted: April 2016

Abstract Cloud data centres have become the preferred hosting environment for large scale web-facing applications. They allow unprecedented scalability in response to a dynamic and unpredictable workload. However, many applications need to scale beyond the boundaries of a single data centre to multiple geographically dispersed clouds sites (i.e. a Multi-Cloud) to provide regulatory compliance, better Quality of Experience (QoE) and increased fault tolerance. In this work, we introduce a flexible framework which allows interactive web applications to utilise a Multi-Cloud environment. It redirects users to suitable cloud sites considering the latency and regulatory constraints. Regulatory requirements are specified via a flexible and simple domain specific model, which is then interpreted by a rule inference engine. We conducted an experimental evaluation of the proposed system using services of 10 cloud sites/data centres located in 5 continents and offered by two cloud providers, namely Amazon and NeCTAR. The results show that our approach minimises latency, is fault tolerant, and meets all stated regulatory requirements with negligible performance overhead.

Keywords Cloud Computing · Multi-Cloud · Web applications

1 Introduction

Cloud computing is a disruptive model of leasing and using IT resources. It allows organisations to concentrate on their main line of business without invest-

N. Grozev
Cloud Computing and Distributed Systems (CLOUDS) Laboratory
Department of Computing and Information Systems, The University of Melbourne, Australia
E-mail: nikolay.grozev@unimelb.edu.au

R. Buyya
Cloud Computing and Distributed Systems (CLOUDS) Laboratory
Department of Computing and Information Systems, The University of Melbourne, Australia
E-mail: rbuyya@unimelb.edu.au

ing in their own private computing infrastructure. Using an Infrastructure as a Service (IaaS) cloud offering, enterprises can rent a dynamic pool of preconfigured computational and storage resources in a *pay-as-you-go* manner [14, 31]. Hence, they can dynamically resize/scale their rented virtual infrastructure and avoid investing in a fixed resource pool, which can often be either under or over-utilised depending on the workload. Such flexibility is of paramount importance for large scale web-facing applications whose user base can grow or shrink dynamically as a result of product releases, marketing campaigns, holiday seasons etc. Most such applications follow the standard 3-tier architectural pattern and are comprised of three layers/tiers [1, 21, 44]:

- **Presentation Layer** — the interface, used by the end user. Typically, executed in a browser or a mobile device application.
- **Business/Domain Layer** — the core business logic. Hosted in one or several Application Servers (AS).
- **Data Layer** — manages the access to the persistent data. Deployed in one or several Database (DB) servers.

The traditional model of using an IaaS cloud service is to allocate resources (i.e. AS and DB servers) exclusively within a single cloud site. This poses several new barriers to cloud adoption. A data centre outage can leave end users without access to an essential service for long periods of time, as exemplified by several recent incidents [2, 3, 23]. In fact, according to a Berkeley's analysis service unavailability is the greatest obstacle for cloud adoption [9]. Furthermore, the Quality of Experience (QoE) of interactive web-facing applications is highly impacted by the network latency between the end customers and the data centre. If users are distributed worldwide, a single data centre will not be able to provide low latency and acceptable response time to everyone. This has prompted companies like *Ebay* and *IBM* to use multiple data centres to respond to dynamic workloads and increase end user QoE [18, 27].

As many businesses operate across national borders, they have to consider a plethora of legislative and regulatory constraints with respect to privacy, data access control, security, etc. Government and academic reports have outlined the complexities of building regulatory compliant cloud services and emphasised the need for engineering approaches enabling such compliance [10, 11, 39, 40]. As an example, the Data Protection Directive of the European Union (EU) forbids the transfer of personal data to non-member countries, unless an adequate level of protection is ensured [19]. Similar laws exist in many other jurisdictions as well [11]. Businesses operating in many legislative domains have to meet multiple and often conflicting regulatory requirements and thus no single cloud site will suffice.

To address these problems, the usage of multiple cloud sites (i.e. a Multi-Cloud) has gained significant interest. A Multi-Cloud is a type of Inter-Cloud, where clients use resources from multiple cloud sites without relying on their interoperability [20, 25, 41]. A Multi-Cloud environment can provide increased availability. If a cloud site fails, end users can be redirected to alternate locations. Multiple geographically distributed cloud sites can serve users worldwide with low network latency and thus provide adequate QoE. If an application has specific requirements as to where (e.g. in which jurisdiction) a user is served, this user can be redirected accordingly to one of the employed sites.

In this work, we propose a system for load distribution of web facing interactive cloud applications. We detail its underlying design and algorithms, describe the technical aspects of our prototype, and experiment with it in real life cloud environments. More specifically, our key **contributions** are: (i) design and implementation of architectural components for regulations and latency aware user redirection in a Multi-Cloud environment and (ii) a domain specific approach for defining regulatory constrains.

The rest of the paper is organised as follows: In Section 2 we provide an overview of the related works and compare them to ours. In Section 3 we outline the architectural components and their interaction model. Section 4 presents a detailed description of all system components' implementation in terms of used technologies, algorithms, and configurations. It also proposes a domain specific approach for defining regulatory constrains. Our experimental settings and results are discussed in Section 5. In the final Section 6, we conclude the paper and define pathways for future work.

2 Related Work

Our approach encompasses load distribution and resource selection in a Multi-Cloud and considers the regulatory requirements. We discuss the related work in these fields separately.

2.1 Multi-Cloud Resource Selection and Provisioning

Resource utilisation from multiple clouds has already attracted notable interest, signified by at least 20 related projects from both academia and industry [25]. Projects like OPTIMIS [20], Contrail [15], mOSAIC [42], MODAClouds [8], STRATOS [37], and the work of Liu et al. [29] can dynamically select and provision resources in multiple clouds. More recent work has focused on Quality of Service (QoS) prediction and selection of web services, which could be hosted in a distributed environment (e.g. in different clouds) [30], [46]. However, they do not consider the geographical locations of the cloud sites they select from. Hence, they can not implement regulations aware application brokering. Moreover, these projects focus on resource allocation and do not implement workload (i.e. end user) redirection among the cloud sites. In contrast, our approach focuses on regulation aware load distribution.

Significant work has been done in the area of scaling the persistent layer across servers and ultimately across data centres. Cattel [16] surveys more than 20 projects in the area of scalable distributed data bases that balance differently between the consistency, availability and partition tolerance requirements as defined in the CAP theorem [12, 13]. Furthermore, Google has revealed their database architecture which scales within and across geographically distributed data centres without violating transaction consistency [17]. Given how well explored this area is, persistent data distribution across data centres is outside the scope of this work. We focus on load balancing the incoming users across data centres given that the data is already appropriately distributed across the cloud sites.

In our previous work, we introduced mechanisms for efficient 3-tier application brokering in multiple clouds [26]. We addressed load distribution and resource scheduling both across and within cloud sites and carried performance evaluation through discrete event simulations. In this work, we focus on load distribution across data centres only. Instead of using discrete event simulations we develop a real life prototype, which we use for evaluation as well. We detail the technical aspects of our implementation and optimisation techniques that allow our approach to scale adequately under heavy load. Also, we explore in details how regulatory requirements can be flexibly specified with minimal efforts in a domain specific model and efficiently interpreted at runtime. Finally, we have extended our network latency estimation approach to achieve better accuracy and scalability.

2.2 Load Distribution in Industry

Figure 1 shows how a 3-tier application can be deployed in a Multi-Cloud using the industry best practices. Each user must be redirected among the cloud sites. This can be achieved through services like AWS Route 53 [4] and AWS Elastic Load Balancer (ELB) [5]. Typically, AWS ELB is used to load balance across AS Virtual Machines (VMs) within a single cloud site. It can also distribute requests among a number of AWS availability zones, but can not use resources from other providers. Route 53 is a Domain Name System (DNS) service and provides Latency Based Routing (LBR), which redirects each user to the cloud site providing the lowest latency to him/her.

Once a user is redirected to a cloud site, he/she is served within that site and has no further interaction with the Route 53 service. That is, they are served by the AS and DB servers within the selected cloud site only. Similarly to Route 53, we also attempt to minimise the latency between the end users and the destination cloud sites. However, unlike ELB and Route 53 we also consider the regulatory requirements and the end user's identity.

Content Delivery Networks (CDN) have some resemblance with Route 53 and our approach because they efficiently deliver web content hosted in multiple data centres to users distributed worldwide. However, a CDN only delivers static web content, while we direct users to cloud sites where they are interactively served with dynamic web content.

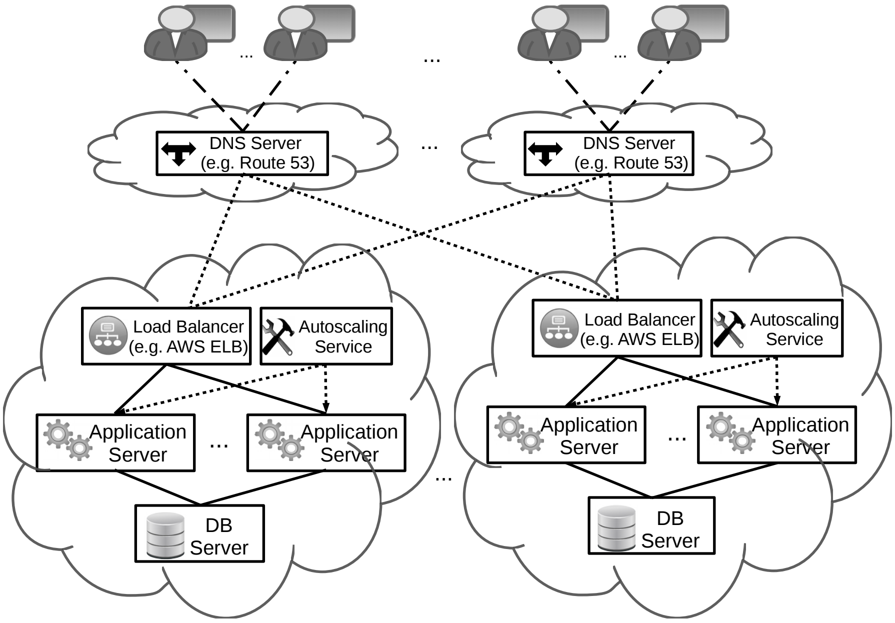

Fig. 1 Current practice for deploying 3-tier applications in a Multi-Cloud.

2.3 Specifying and Enforcing Regulations

Mont et al. proposed an approach allowing end users to specify their own data protection rules [32]. They also introduced a method for service providers to enforce such rules by using sticky policies. Their work was later prototyped by Mowbray et al. [33]. This model is not applicable when a cloud provider does not collaborate to enforce the stated requirements, while our approach is cloud agnostic. Furthermore, our approach works in a Multi-Cloud environment, while theirs is applicable only within a single cloud site.

Several formalisms for specifying access policies have been proposed. Two notable examples are OASIS XACML [34] and EPAL [28]. Our approach uses a rule based inference engine to infer the suitability of a cloud site for a user. Therefore, we found it convenient to express the access policies directly in the declarative language of the rule engine. As a future work, we are planning to investigate how XACML and EPAL specifications can be transformed to such engine rules.

In order to dispatch a user to an appropriate cloud site, we first need to authenticate him/her, which can be done in several ways. The OpenID protocol [36] allows a service provider to authenticate an end user through an external service. The OAuth [35] protocol allows a service provider to gain authorised access to protected user resources hosted on another service without explicitly providing credentials (e.g. user name/password). Alike OpenID, OAuth can be used for authentication as well, if the accessed resources are

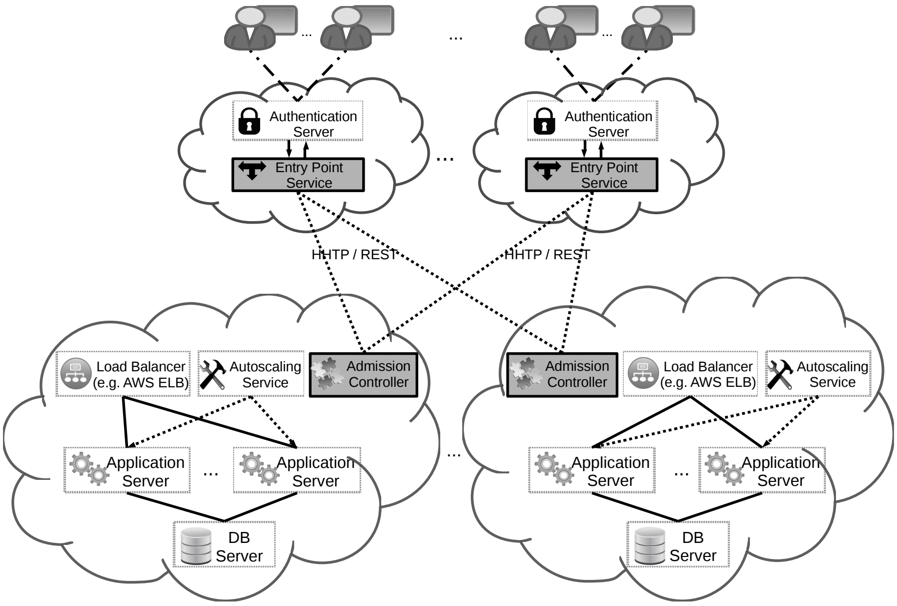

Fig. 2 Overall Architecture: *Entry Point* and *Admission Controller* components replace the DNS based cloud site selection.

only user metadata (e.g. user name). More and more web services are adopting OpenID and OAuth. However, many others still rely on custom authentication (e.g. user name/password). Since we aim for generality, we decided not to couple our approach with any of these specifications. Instead, our approach works with user tokens, which are unique ids and can be extracted from OpenID, OAuth or via custom authentication.

3 System Architecture

Our approach to cloud site selection (depicted on Figure 2) extends the aforementioned industry standard approach by replacing the DNS service with an *Entry Point* component. Similarly, it redirects users among a set of cloud locations. Furthermore, within each target cloud site we deploy an additional *Admission Controller* component. *Entry Points* communicate with the *Admission Controllers* to determine the suitability of the respective cloud sites and then redirect the user accordingly. Apart from these two new components, there are no other changes to the industry standard approach. Therefore, in each cloud the system can be implemented in accordance with the reference 3-tier approach. In other words, what we propose is a layer “on top” of the standard 3-tier architecture, which performs the user redirection to the individual cloud sites.

An end user arrives at an *Entry Point*. There may be one or more *Entry Points* whose purpose is to redirect each user to an appropriate cloud site.

Before doing so, the user must authenticate so the regulatory requirements can be considered. Authentication is application specific and is provided by the application developers. It could be OAuth or OpenID authentication with an external service, or a custom username/password implementation. Once the Authentication is complete, the *Authentication Server* passes a unique user identifier (e.g. retrieved from an OAuth server or just a user name) to the *Entry Point.*

The *Entry Point* can be called in two ways. Since it is developed in Java, it can be put on the java classpath of the *Authentication Server.* It can also be accessed as a RESTful web service in case the *Authentication Server* is not developed in Java. An *Entry Point* is deployed in a separate VM. Depending on its implementation, the authentication service could be deployed within the same VM or separately.

In each target cloud site there is an *Admission Controller* which exposes a RESTful web service API and is deployed in a standalone web server. The *Entry Point* sends a request to each cloud site's *Admission Controller* to determine which sites are eligible in terms of regulations to serve the incoming users. We have developed a domain specific rule based model (detailed in Section 4.2) allowing for succinct regulations requirements specification. Based on this information and estimations of the network latency between the end user and each cloud site, the *Entry Point* decides where to redirect the user to. Figure 3 depicts this flow.

Once the user is redirected to a cloud site, he/she is served there and has no further interaction with the *Entry Points* and the *Admission Controllers.* In other words, the delay resulting from the interactions between the *Entry Point* and the *Admission Controller* is tangible to the user only during the initial redirection process and does not impede the overall QoE. We argue that existing web-facing systems, which use OAuth 2.0 authentication with external services (e.g. Google, Facebook or LinkedIn) already serve users with one-off initial delays, and thus this is acceptable.

Although placing the *Admission Controllers* in all cloud sites contributes to the one-off delay, it allows them to closely monitor the resource performance and utilisation within each cloud site. This information can be fed back to the *Entry Points* to implement sophisticated redirection policies. We are planning to investigate this in our future work and thus we aim for a flexible architecture.

4 Design and Implementation

4.1 Entry Points

The function of each *Entry Point* is to redirect each incoming user to an appropriate cloud site. After a user is redirected, he/she has no further interaction with the *Entry Point.* Thus, it introduces a one-off delay upon user arrival and does not affect the subsequent user interactions.

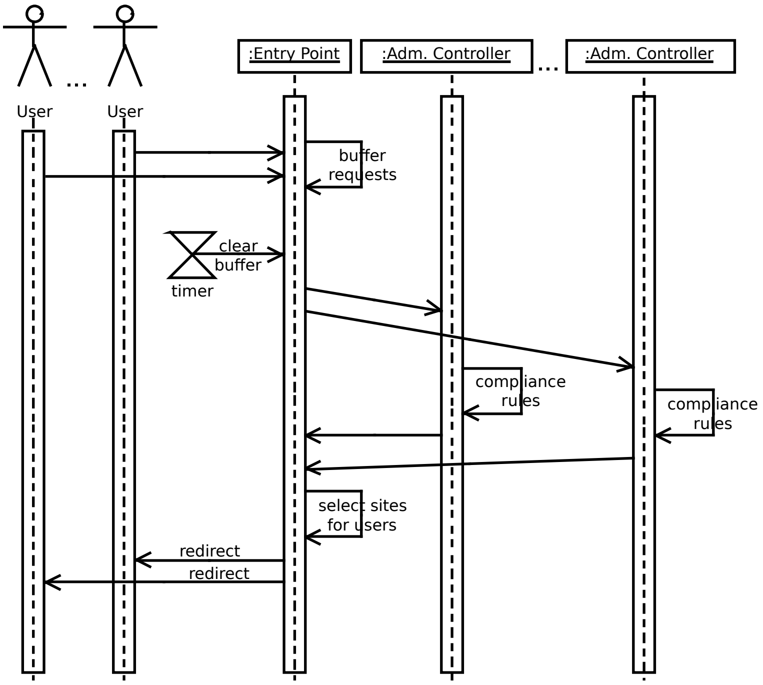

Fig. 3 *Entry Point - Admission Controller* interaction.

Each *Entry Point* is developed in Java and is deployed into a single Java archive (i.e. *jar*) file. It can be used from another Java application by adding the jar file to its classpath. To simplify the integration with the *Authentication Service*, the jar exposes a singleton façade class called *EntryPoint*, which for a given user determines the serving cloud site.

The *Entry Point* can be executed separately as a standalone application as well. In this case, an embedded *jetty* web server is started, which uses the *Jersey* RESTful Web Services framework to expose a REST web service. This is useful when the client code is not easily integrated with Java. Details on how this is achieved can be found in our online documentation [1]. Client code can call this service by providing the user identification token and IP address and receive as output the address of the *Load Balancer* in the selected cloud site. Behind the scenes, the RESTful web service simply calls the aforementioned *Entry Point* façade.

Figure 3 illustrates the interactions between the *Entry Point*, the users, and the *Admission Controllers* which is discussed hereafter. For brevity it omits the user authentication.

To avoid excessive network communication, the *Entry Point* sends the incoming user requests to the *Admission Controllers* in batch rather than one by

[1] https://nikolaygrozev.wordpress.com/2014/10/16/rest-with-embedded-jetty-and-jersey-in-a-single-jar-step-by-step/

one. The *Entry Point* aggregates the user requests over a time period (typically a few seconds) and sends them to the *Admission Controllers* at once. This is transparent from the caller's perspective, as the procedure call is blocked until the result for the respective user request is complete. Furthermore, requests are broadcasted to the *Admission Controllers* asynchronously. A separate thread from a cached thread pool is used for each *Admission Controller* in order to reduce the overall response time.

We emphasise that the delays during the initial end user redirection are one-off. After the user is redirected to a cloud site they only communicate with its load balancer. There are no additional delays in the user's interaction with the application. Furthermore in Section 5 we demonstrate experimentally that these initial delays are negligible.

In the case of a cloud site failure, the respective *Admission Controller* will not respond to the *Entry Point* requests. Hence, the *Entry Points* are preconfigured with deadlines and if a cloud site does not respond within the deadline it is not considered. If however, this *Admission Controller* "reappears" online — the *Entry Point* will start considering it again. For this purpose, each *Entry Point* is configured with a time period parameter, so that it can check if failed *Admission Controllers* have reappeared online. Once the responses from the *Admission Controllers* have been received, the *Entry Point* selects the eligible cloud site, which offers the lowest network latency.

We developed a utility which estimates the network latency between an end user and a cloud site. Given two arbitrary IP address (e.g. of the user and the load balancer in a cloud site) it estimates the expected network latency between them. We approach this problem in two steps. Firstly, we use the MaxMind's GeoLite [22] database to determine the location (i.e. longitude and latitude) of each IP address. The latest GeoLite database is freely available and can be downloaded in a proprietary format. MaxMind also provides an open source Java library to read and query the proprietary database, which we use.

MaxMind's GeoLite is updated once every few months and thus it may not resolve some IP addresses' respective locations correctly. This can often happen for IP addresses of cloud VMs, as a cloud provider can dynamically reassign IPs from the same range to machines in different data locations. Experimenting with GeoLite we found that while end users' IP addresses are mostly resolved accurately, the estimated location of AWS EC2 VMs based on their public IP address can be very incorrect. For example, a VM in the Tokyo AWS region can be resolved as located in the US. Fortunately, cloud providers like Amazon AWS provide up to date information about their IP ranges and the respective geographical locations [6]. In case an IP address falls within any of these ranges we can use it to override the respective value of GeoLite.

Secondly, once the coordinates are established, we use PingER which is an Internet Performance Monitoring Service (IEPM) [43]. It records network characteristics (e.g. round trip time) between hundreds of nodes distributed worldwide which periodically ping each other. The reported performance metrics can be averaged per time interval (e.g. year, month, day or hour) and can be downloaded in tab separated values (TSV) files through a public web service. The coordinates (longitude and latitude) of each node are provided as well. We use Vincenty's formulae to compute the distance between any two geospatial positions. To estimate the latency between two IP addresses, whose positions have already been resolved, we use the network latencies between the 3 geometrically closest to them pairs of PingER nodes.

More formally, for each pair of PingER nodes (n_{i1}, n_{i2}) we define its distance to the locations of the target pair of IP addresses (t_1, t_2) as:

$$\begin{aligned} distance(n_{i1}, n_{i2}, t_1, t_2) = \\ \min\{v(n_{i1}, t_1) + v(n_{i2}, t_2), v(n_{i1}, t_2) + v(n_{i2}, t_1)\} \end{aligned} \quad (1)$$

where v is the well known Vincenty's function for computing the distance between two geographical locations specified by their coordinates. Then we choose the 3 pairs of nodes $(n_{11}, n_{12}), (n_{21}, n_{22}), (n_{31}, n_{32})$, which minimise the *distance* function for the targeted t_1 and t_2. Hence, we can approximate the Internet latency between t_1 and t_2 as the following weighted sum:

$$latency(t_1, t_2) = (\sum \frac{d_{min}.l_i}{d_i})/(\sum \frac{d_{min}}{d_i}) \quad (2)$$

where d_i is a shorthand for $distance(n_{i1}, n_{i2}, t_1, t_2)$, d_{min} is the smallest distance among d_1, d_2, d_3 and l_i is the PingER latency between n_{i1} and n_{i2}. Eq. 2 essentially defines a weighted sum of the network latencies between the 3 geometrically closest pairs of nodes. The weights are defined proportionally to each pair's nodes combined distance to the target locations.

To improve the performance of the network latency estimation we keep in-memory *least-recently-used* (LRU) caches of (i) the mappings between IP addresses and coordinates, (ii) the distances between geospatial locations and (iii) the already resolved latencies. To improve the cache-hit probability, we round all coordinates (latitude and latency) to 2 digits after the decimal sign when comparing with or inserting into the caches. This ensures the network latency estimation for users coming from the same region (e.g. the same city) will not be recomputed every time. We use the Google Guava [24] cache implementation as it is thread safe, and limit all cache sizes to 10 000 entries.

Finally, in the previous discussion we mentioned several configuration parameters used by the *Entry Point*. These are provided in two configuration files. The first one is a comma separated values (CSV) file which lists all cloud sites and the addresses of their *Admission Controllers* and *Load Balancers*, from which the load balancer chooses when a user arrives. The second file is a property file, and provides (i) the time between the batch requests to the *Admission Controllers* and (ii) the deadline length to wait for response from the *Admission Controllers*. These files must be provided before the *Entry Point Service* is used, but can also be updated and reloaded dynamically to change its behaviour - e.g. to add a new cloud site.

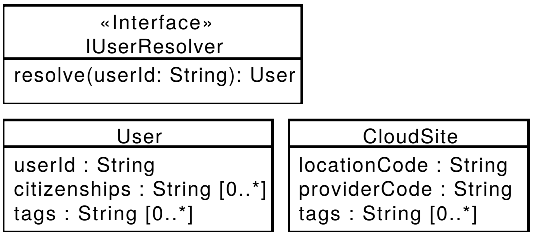

Fig. 4 Domain model classes used for cloud site to user matching.

4.2 Admission Controllers

Each *Admission Controller* is deployed in a single jar file and is executed within an embedded jetty server hosting a RESTful web service. The service takes as input a list of user identifiers, and returns the responses as a list of JSON formatted objects. The *Entry Points* communicate with the *Admission Controllers* through this web service, even if they are deployed in the same VM.

The *Admission Controller* must determine the eligibility of the cloud site for each of the users. To achieve this, first we must resolve the user's metadata — e.g. data about their citizenships and privacy requirements. Obviously this resolution is application specific. For example, one application would use OAuth to retrieve such data, while another would just execute a query within its data base. Since we aim for generality, our *Admission Controller* component provides a "hook" for resolving user metadata. We define a simple interface called *IUserResolver*, which has a single method for resolving a user's metadata. Application developers should implement this interface and provide the name of their class as an input parameter to the *Admission controller*.

We also define two simple classes representing a user and a cloud site. The *User* class has just 3 fields: (i) user id, (ii) set of citizenships and (iii) tags. The *tags* field can aggregate miscellaneous additional meta-information for the user - e.g. if he/she is a government employee. The *Admission Controller* uses the previously discussed user resolver to instantiate this class based on a user id. The *CloudSite* class represents meta-information about the present cloud site. It has fields representing: (i) the geographical location (e.g. country or state code) (ii) the cloud provider (e.g. AWS) and (iii) a set of tags. As before, we use tags to represent miscellaneous additional information - e.g. if the cloud is certified to serve US government agencies. A JSON file containing the respective marshalled *CloudSite* instance is given as a parameter to every *Admission Controller*.

The *User*, *IUserResolver* and *CloudSite* types (depicted on Figure 4) define the simple interface, which we expose to application programmers.

Given a user and a cloud site, the *Admission Controller* must infer if the user could be served there. An example might help picture what the reasoning

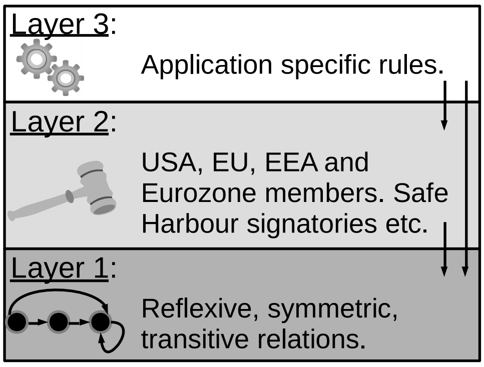
Layer 3:
Application specific rules.
Layer 2:
USA, EU, EEA and Eurozone members. Safe Harbour signatories etc.
Layer 1:
Reflexive, symmetric, transitive relations.

Fig. 5 Layers of DRL rules. Each layer can use the rules from the underneath layers.

logic should be. Let us consider an application which needs to ensure that EU citizens are served within the EU, and that government officials must be served in specifically certified cloud sites. Given a user id, we can resolve the user's meta-information with the provided *IUserResolver*. Let us assume this returns a user with French citizenship, who is also a government official (denoted by a tag in the *User* instances). Let's also assume the corresponding cloud site (represented with a *CloudSite* instance) is located in Germany and the provider is AWS. We may also know that all AWS cloud sites comply with the government certification in question. In this case we should be able to infer that the cloud site is eligible to serve the client.

This type of automated reasoning based on predefined facts and rules is a hallmark application of rule based engines and this is exactly how we approach this problem. In our implementation of the *Admission Controller* we use the Drools rules management system [45]. Drools uses the ReteOO algorithm, which implements well known performance enhancements of the famous Rete algorithm. Drools uses a domain specific language called Drools Rule Language (DRL) to define rules and facts. It integrates well with existing Java code and facts in DRL are just Java objects. This allows the declarative programming approach prescribed by DRL to also make use of the polymorphic features of Java. For example, if a rule is activated for any fact of a specific Java type, it will also be activated for any instance of its subtypes. We will make use of this feature to allow more flexible specification of regulatory rules.

Specifying compliance rules in a declarative DRL form rather than procedurally is beneficial in terms of maintainability and flexibility. However, this still needs to be done on a per-application basis. Many applications would share common rules and facts — e.g. about which countries participate in the U.S.-EU Safe Harbour agreement. It would be beneficial if we specify such common rules so they can be reused by applications. Hence, we divide the rules in 3 layers/modules depicted in Figure 5. The rules from each layer can only use the roles from the layers below.

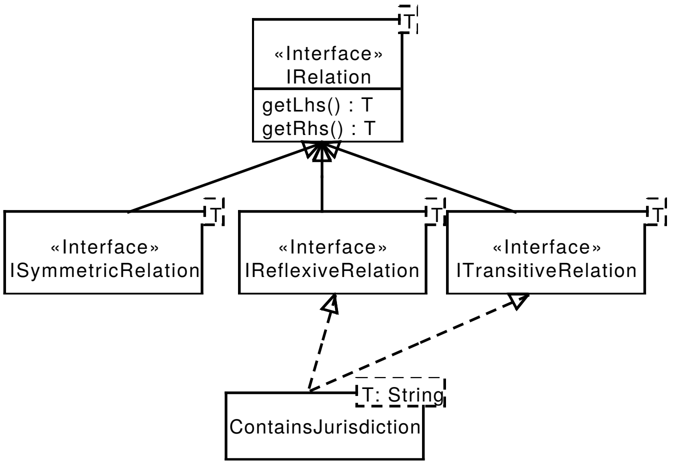

Fig. 6 Interfaces for the relations from Layer 1 and the *ContainsJurisdiction* relation from Layer 2.

In Layer 1 we model the concepts of reflexive, symmetric, and transitive binary relations. We introduce 3 Java interfaces to represent these concepts. They all extend from a common *IRelation* interface which has a left and a right hand side (*lhs* and *rhs*) objects — see Figure 6. Layer 1 includes DRL rules, which produce new *IRelation* instances in the Drools working memory based on the reflexive, symmetric and transitive algebraic rules. For example, if we have an instance of *ISymmetricRelation*, whose $lhs = a$ and $rhs = b$, the respective rule in Layer 1 will be activated and will insert in working memory a relation of the same type, whose $lhs = b$ and $rhs = a$.

To illustrate how this is useful, let us consider how we can implement a relation of jurisdictional containment. We can define a relation *ContainsJurisdiction*, which implements *IReflexiveRelation* and *ITransitiveRelation*. Then we can use it to specify the facts that Germany is within the Eurozone, which in turn is in the EU. The rules in Layer 1 will automatically fire and populate the Drools working memory with the appropriate *ContainsJurisdiction* instances denoting that Germany is in the EU and itself. This simplifies significantly the development of rules using jurisdictions. Figure 6 depicts the aforementioned interfaces and class.

In Layer 2 we specify common facts and rules about the regulatory domain and cloud providers. In fact the aforementioned *ContainsJurisdiction* relation is defined and extensively used in Layer 2, to specify which countries are in the Eurozone, EU and The European Economic Area (EEA), which countries have signed the US-EU Safe Harbour agreement and so on. In essence, this layer contains the expert knowledge about the regulatory domain in the form of DRL rules.

Finally, using the domain rules from Layer 2 we can implement the actual admission control policies in Layer 3, which are application specific. We define a class *AdmissionDenied*, which has a single property — the unique user id. When the *Admission Controller* starts the DRL rules, it inserts into the working memory the meta-information of one or more users and the cloud site as instances of the aforementioned types. Based on this, the rules from Layer 3, should insert an *AdmissionDenied* instance in working memory for every user who is not eligible for the specified cloud site.

```
rule ”US government officials – in US only”
when
    User($id: userId, tags contains ”US–GOV”)
    CloudSite($lc : locationCode)
    not(ContainsJurisdiction(lhs == ”USA”, rhs == $lc))
then
    insert (new AdmissionDenied($id));
end
```

Listing 1 Sample Layer 3 admission control rule

As an example, Listing 1 demonstrates how we can restrict the redirection of US government official to cloud sites outside of the US. Government officials are recognised by a tag in the respective *User* instance.

In Drools one can specify the *salience* of each rule. This is an integer used by the Drools engine when deciding the order or rule execution. We give the rules in Layer 1 highest salience (200) and we give lower salience (100) to the rules from Layer 2. The rules in Layer 3 get the lowest salience of 0. This causes the rules from the lower layers to execute with higher priority (i.e. earlier if possible). Thus when the higher layer rules are run the underlying knowledge base is already populated.

The logical separation of rules into layers helps in the change management as well. The base rules from Layer 1 should not change for any applications. The rules from Layer 2 should rarely change — for example, when a new country joins the EU, or when new regulation is enforced. The rules from Layer 3 may change frequently and are application specific. Drools allows developers to specify rules in separate files. We use this feature to separate the layers in 3 different files. Application developers need to provide only the file for Layer 3.

5 Performance Evaluation

To evaluate our approach we perform two experiments in a heterogeneous Multi-Cloud environment. We employ the following cloud sites:

- AWS Region in Oregon, US;
- AWS Region in Northern California, US;
- AWS Region in Northern Virginia, US;
- AWS Region in São Paulo, Brasil;
- AWS Region in Dublin, Ireland;
- AWS Region in Frankfurt, Germany;

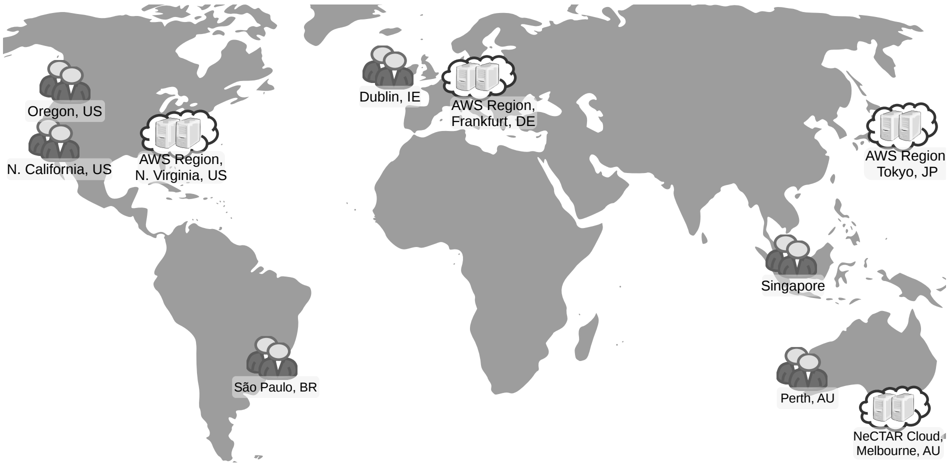

Fig. 7 Experiment 1: 4 cloud sites host the *Entry Points* and *Admission Controllers*; 6 cloud sites are used to emulate incoming users.

- AWS Region in Tokyo, Japan;
- AWS Region in Singapore;
- NeCTAR data centre in Perth, Australia;
- NeCTAR data centre in Melbourne, Australia.

In both experiments we deploy *Admission Controllers* and *Entry Points* in multiple cloud sites around the world. Then we use other geographically distributed cloud sites to emulate incoming users. We record multiple performance characteristics to demonstrate the viability of our approach under strenuous load.

5.1 Experiment 1 - Large Scale Deployment

The goal of our first experiment is to show that our workload distribution system (i) adds negligible delay when users access the system (ii) honours all regulatory requirements and (iii) can handle significant workloads without requiring a lot of resources.

In each of 4 different cloud sites we deploy an *Entry Point* and an *Admission Controller* in a shared VM. We use *m3.medium* instances in the N. Virginia, Frankfurt and Tokyo AWS regions and a *m2.medium* instance in the Melbourne data centre of the NeCTAR Research Cloud. To emulate users from across the globe we use *m3.medium* instances provisioned in the Oregon, N. California, São Paulo, Dublin and Singapore AWS regions and a *m2.medium* instance in the Western Australian site of NeCTAR near Perth. Figure 7 depicts the locations we use for Experiment 1. All VMs run Ubuntu 14.04 and we have increased their maximal open files and socket connection limits, so they can handle more connections.

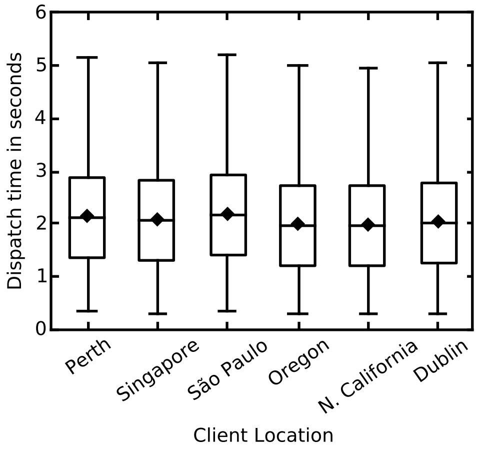

Fig. 8 Dispatch times of users to serving data cites, grouped by location of the emulated users. Mean values are denoted with a rhombus.

The first experiment has been conducted for the duration of 24 hours. To test how the system behaves under significant load, we emulate 2 users per second from each of the 6 cloud sites we use to represent clients. This amounts to more than 1 million emulated users during the experiment altogether. Each emulated user connects randomly to one of the 4 *Entry Points* and provides a random user id. In the *Entry Points*, we need to resolve this id to a *User* instance. We do so by implementing a custom *IUserResolver* instance. Given a user id it creates a *User* instance by randomly assigning his/her citizenship to one of the following countries: Germany, USA, Australia and Canada. To every third user with US citizenship we assign the tag “US-GOV” denoting that they are government officials. To every second user we assign the “PCI-DSS” tag. The Payment Card Industry Data Security Standard (PCI DSS) formalises procedures for reducing credit card frauds [38]. Tagging a user with “PCI-DSS” means they are allowed to work with credit cards data within the system.

We configure each *Admission Controller* with the geographical location and the provider name of the respective cloud site. All AWS cloud sites are marked with the “PCI-DSS” tag, as AWS is PCI DSS compliant [7]. The NeCTAR cloud sites are not tagged as “PCI-DSS”, as it is lacking official compliance.

We configure the *Admission Controllers* with the following regulatory requirements expressed as Layer 3 DRL rules:

- Users having access to credit card data (i.e. tagged with “PCI-DSS”) should be served in PCI DSS compliant cloud sites.
- EU citizens should be served within the EU.
- US government officials should be served within the US.

Figure 8 shows the distributions of dispatch times of the emulated end users grouped by their location. By “dispatch time” we denote the time from

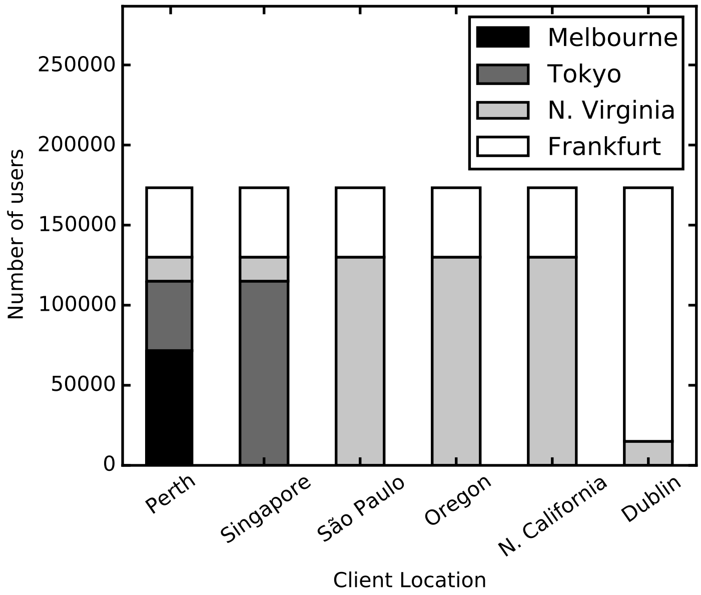

Fig. 9 Number of users redirected to each cloud site grouped by location.

Table 1 Network latencies between clients and cloud sites in milliseconds.

	N.Virginia	Frankfurt	Melbourne	Tokyo
Perth	137.0	180.0	**20.0**	124.0
Singapore	121.0	131.0	69.0	**44.0**
São Paulo	**61.0**	106.0	224.0	140.0
Oregon	**38.0**	73.0	88.0	45.0
N.Cal.	**41.0**	84.0	84.0	52.0
Dublin	39.0	**11.0**	159.0	108.0

the user emitting a request to an *Entry Point* until he/she is redirected to an appropriate cloud site — i.e. the duration of the procedure depicted in Figure 3. This delay is experienced just once, before the user is redirected to the destination cloud site. Regardless of the user location the mean and median dispatch times are around 2 seconds and the majority of values are less than 3 seconds. The mean and median for the users from São Paulo are slightly higher (with approximately 0.18 seconds) because of the higher latencies to all other cloud sites.

Table 1 lists the actual network latencies between the emulated clients and the cloud sites. The best latencies are highlighted. These values are obtained by performing 50 measurements between each pair and then taking the median. Ideally, for a given user our approach should prefer the cloud site with the lowest latency unless there are specific regulatory requirements for him/her.

Figure 9 shows the proportion of clients from each location dispatched to the different cloud sites. As per Table 1 the cloud site in North Virginia offers the best latency to the clients in São Paulo, Oregon and California, and thus 75% of the users from these locations are redirected there. The other 25% are redirected to Frankfurt in accordance with the regulatory requirements, as they are German citizens. Most users from Dublin are redirected to the Frankfurt site, except to those (approx. 8.3%) who are tagged with "US-GOV" and must be served in the US. Likewise, most users from Singapore were redirected to Tokyo, except for EU citizens and US government officials.

Similarly, 25% of the users from Perth were directed to Frankfurt, and 8.3% to N. Virginia in order to meet the aforementioned requirements. However, only half of the rest were directed to the Melbourne site, although it offers the lowest latency. This is because the NeCTAR cloud is not PCI DSS compliant and thus users working with credit cards have to be served elsewhere. Such users are directed to the Tokyo AWS cloud site, as it offers the lowest latency from all PCI DSS compliant sites.

5.2 Experiment 2 - Fault Tolerance

The goal of our second experiment is to show how the system behaves when a cloud site fails. In this experiment we deploy a single *Entry Point* in a separate *m3.medium* VM in the São Paulo AWS site. Also, we deploy two *Admission Controllers* in *m3.medium* instances in the N. Virginia and Frankfurt sites. Alike the previous experiment we emulate users from VMs in the Dublin and Oregon data centres. The frequencies with which users are started is the same as in Experiment 1. However, in this experiment we do not configure any regulatory rules — our goal is to test failure tolerance. The length of Experiment 2 is 3 hours.

To simulate failure every 20 minutes we switch off one of the *Admission Controllers* for 10 minutes, after which we start it again. We switch off the two *Admission Controllers* in turns. Figure 10 shows the proportion of the overall clients directed to each cloud site during the 10 minute periods. As per Table 1 the Frankfurt site offers better latency for users in Dublin and the N.Virginia site for those in Oregon. Hence, during the 10 minute periods when both *Admission Controllers* were present the two cloud sites receive about equal number of users, as we emulate equal number of users from Oregon and Dublin. During the periods when one of the *Admission Controllers* is not present all users are directed to the other one.

From Figure 10 we can see that during the 10 minute periods after an *Admission Controller* is started again it can receive slightly less than the expected 50% of users. In our experiments we have configured the *Entry Points* to check if disconnected *Admission Controllers* have come back online every 1 minute — this is a configurable parameter. Hence, the *Entry Point* can be oblivious that an *Admission Controller* is back online for up to 1 minute, causing all incoming users to be redirected to the alternative cloud site.

Figure 11 depicts the dispatch time during each 30 minutes of Experiment 2. It shows that throughout the experiment the mean and median dispatch

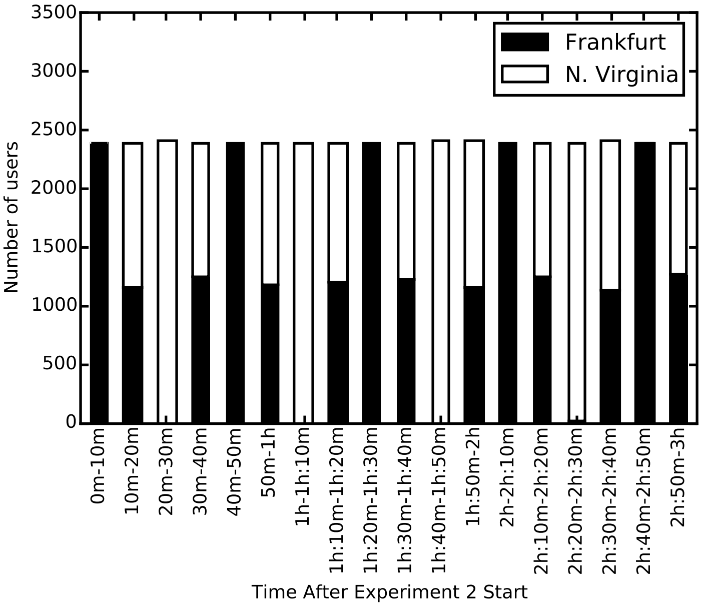

Fig. 10 Experiment 2: Number of users directed to each cloud site. In this experiment, the *Admission Controllers* are one-by-one switched off for 10 minutes and then started again for 10 minutes. The first *Admission Controller* to be shut down is the one in North Virginia.

times remain below 3 seconds. We have configured the *Entry Point* to connect to the *Admission Controllers* with a timeout of 10 seconds. This parameter is configurable as well. If the connection fails the respective cloud site is marked as failed, and is not used again until it reappears back online, as explained previously. Hence, users arriving at the *Entry Point* immediately after an *Admission Controller* has been shut down can experience a dispatch delay of up to 14 seconds, which is nearly 10 seconds more than the 3rd quartile. This is represented by the outliers in Figure 11.

6 Conclusions and Future Work

In this work, we have introduced an approach for distributing end users of an interactive web application across multiple cloud sites. We described a rule based domain specific model, which allows regulatory requirements to be specified with minimal efforts. Our system ensures that these requirements are honoured. We provide extension points or “hooks” that allow developers to use their custom authentication and user details resolution mechanisms. Furthermore, our approach estimates the potential network latencies to each cloud sites and chooses accordingly. We deployed our prototype in multiple cloud sites worldwide and carried extensive experiments showing that indeed

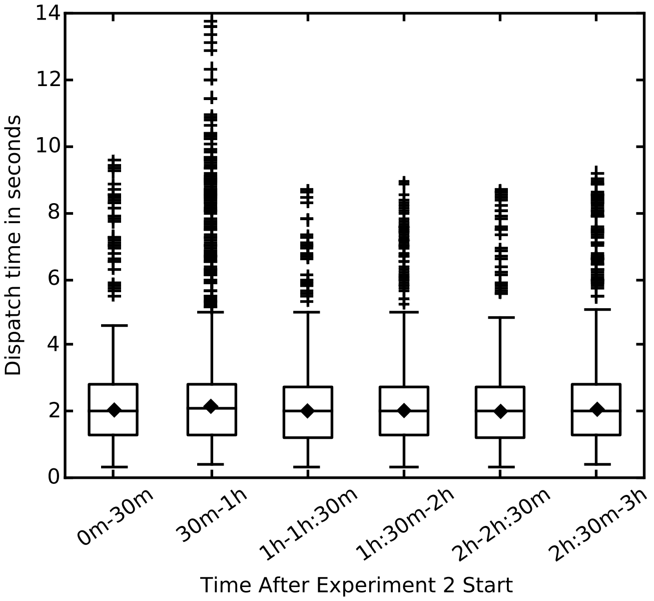

Fig. 11 Experiment 2: Dispatch times over time.

it is fault tolerant, has negligible performance overhead, minimises latency, and meets the stated regulatory requirements.

In the future we will work on incorporating cost and performance knowledge into our cloud selection method. Furthermore, we will investigate how to convert security policies expressed in standards like OASIS XACM and EPAL to DRL regulatory rules.

Acknowledgements

We thank Rodrigo Calheiros, Amir Vahid Dastjerdi, Adel Nadjaran Toosi and the rest of the CLOUDS lab members for their comments on improving the paper. We also thank Amazon.com, Inc and the NeCTAR research cloud for providing access to their infrastructure for conducting the experiments reported in this paper.

References

1. Aarsten, A., Brugali, D., Menga, G.: Patterns for three-tier client/server applications. In: Proceedings of Pattern Languages of Programs (PLoP) (1996)
2. Amazon: Summary of the Amazon EC2 and Amazon RDS Service Disruption (2012). URL `http://aws.amazon.com/message/65648/`
3. Amazon: Summary of the AWS Service Event in the US East Region (2012). URL `http://aws.amazon.com/message/67457/`

4. Amazon: Amazon Route 53 (2013). URL `http://aws.amazon.com/route53/`
5. Amazon: Elastic Load Balancing (2013). URL `http://aws.amazon.com/elasticloadbalancing/`
6. Amazon: AWS IP Address Ranges (2014). URL `http://docs.aws.amazon.com/general/latest/gr/aws-ip-ranges.html`
7. Amazon: AWS PCI DSS Level 1 FAQs (2015). URL `http://aws.amazon.com/compliance/pci-dss-level-1-faqs/`
8. Ardagna, D., Di Nitto, E., Mohagheghi, P., Mosser, S., Ballagny, C., D'Andria, F., Casale, G., Matthews, P., Nechifor, C., Petcu, D., Gericke, A., Sheridan, C.: 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), pp. 50–56 (2012)
9. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: A view of cloud computing. Communications of the ACM **53**(4), 50–58 (2010)
10. Australian Government, Department of Communications: Cloud Computing Regulatory Stock Take. Report Version 1 (2014)
11. Bowen, J.A.: Legal issues in cloud computing. In: R. Buyya, J. Broberg, A. Goscinski (eds.) Cloud Computing: Principles and Paradigms, chap. 24, pp. 593–613. Wiley Press (2011)
12. Brewer, E.: Towards Robust Distributed Systems. In: Proceedings of the Annual ACM Symposium on Principles of Distributed Computing, vol. 19, pp. 7–10. ACM, New York, NY, US (2000)
13. Brewer, E.: CAP Twelve Years Later: How the "Rules" Have Changed. Computer **45**(2), 23 (2012)
14. Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation Computer Systems **25**(6), 599–616 (2009)
15. Carlini, E., Coppola, M., Dazzi, P., Ricci, L., Righetti, G.: Cloud Federations in Contrail. In: M.e.a. Alexander (ed.) Proceedings of Euro-Par 2011: Parallel Processing Workshops, *Lecture Notes in Computer Science*, vol. 7155, pp. 159–168. Springer Berlin / Heidelberg, Berlin, Heidelberg (2012)
16. Cattell, R.: Scalable SQL and NoSQL Data Stores. SIGMOD Record **39**(4), 12–27 (2010)
17. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J.J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., Woodford, D.: Spanner: Google's globally distributed database. ACM Transactions on Computer Systems **31**(3), 8:1–8:22 (2013)
18. Ebay: Ebay (2013). URL `http://www.ebay.com/`
19. European Parliament: Data Protection Directive (95/46/EC) (2015). URL `http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:31995L0046:en:HTML`
20. Ferrer, A.J., Hernández, F., Tordsson, J., Elmroth, E., Ali-Eldin, A., Zsigri, C., Sirvent, R., Guitart, J., Badia, R.M., Djemame, K., Ziegler, W., Dimitrakos, T., Nair, S.K., Kousiouris, G., Konstanteli, K., Varvarigou, T., Hudzia, B., Kipp, A., Wesner, S., Corrales, M., Forgó, N., Sharif, T., Sheridan, C.: OPTIMIS: A holistic approach to cloud service provisioning. Future Generation Computer Systems **28**(1), 66–77 (2012)
21. Fowler, M.: Patterns of enterprise application architecture. Addison-Wesley Professional (2003)
22. GeoLite: GeoLite2 Free Downloadable Databases (2013). URL `http://dev.maxmind.com/geoip/legacy/geolite/`
23. Google: Post-mortem for February 24th, 2010 outage (2012). URL `https://groups.google.com/group/google-appengine/browse_thread/thread/a7640a2743922dcf?pli=1`
24. Google: Google Guava (2015). URL `https://github.com/google/guava`
25. Grozev, N., Buyya, R.: Inter-Cloud Architectures and Application Brokering: Taxonomy and Survey. Software: Practice and Experience **44**(3), 369–390 (2012)

26. Grozev, N., Buyya, R.: Multi-Cloud Provisioning and Load Distribution for Three-Tier Applications. ACM Transactions on Autonomous and Adaptive Systems **9**(3), 13:1–13:21 (2014)
27. IBM: IBM takes Australian Open data onto private cloud. Tech. rep., IBM (2013)
28. IBM: Enterprise Privacy Authorization Language (2015). URL `http://www.zurich.ibm.com/security/enterprise-privacy/epal/Specification/index.html`
29. Liu, C.Y., Huang, K.C., Lee, Y.H., Lai, K.C.: Efficient resource allocation mechanism for federated clouds. International Journal of Grid and High Performance Computing **7**(4), 74–87 (2015)
30. Ma, Y., Wang, S., Hung, P.C.K., Hsu, C.H., Sun, Q., Yang, F.: A Highly Accurate Prediction Algorithm for Unknown Web Service QoS Values. IEEE Transactions on Services Computing **PP**(99), 1–14 (2015)
31. Mell, P., Grance, T.: The NIST Definition of Cloud Computing. Special Publication 800-145, National Institute of Standards and Technology (NIST) (2011)
32. Mont, M., Pearson, S., Bramhall, P.: Towards accountable management of identity and privacy: sticky policies and enforceable tracing services. In: Proceedings of the 14th International Workshop on Database and Expert Systems Applications, pp. 377–382 (2003)
33. Mowbray, M., Pearson, S.: A Client-based Privacy Manager for Cloud Computing. In: Proceedings of the 4th International ICST Conference on COMmunication System softWAre and middlewaRE (COMSWARE), pp. 5:1–5:8. ACM, New York, NY, USA (2009)
34. OASIS: eXtensible Access Control Markup Language (XACML) Version 3.0 (2015). URL `http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html`
35. OAuth: OAuth (2015). URL `http://oauth.net/`
36. OpenID Foundation: OpenID (2015). URL `http://openid.net/`
37. Pawluk, P., Simmons, B., Smit, M., Litoiu, M., Mankovski, S.: Introducing STRATOS: A cloud broker service. In: Proceedings of the IEEE International Conference on Cloud Computing (CLOUD). IEEE (2012)
38. PCI Security Standards Council: PCI-DSS (2015). URL `https://www.pcisecuritystandards.org/`
39. Pearson, S.: Taking Account of Privacy when Designing Cloud Computing Services. In: Proceedings of the Workshop on Software Engineering Challenges of Cloud Computing (ICSE), pp. 44–52. IEEE Computer Society, Washington, DC, USA (2009)
40. Pearson, S., Benameur, A.: Privacy, Security and Trust Issues Arising from Cloud Computing. In: Proceedings of the Second IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pp. 693–702 (2010)
41. Petcu, D.: Multi-Cloud: Expectations and Current Approaches. In: Proceedings of the International Workshop on Multi-cloud Applications and Federated Clouds (Multi-Cloud), pp. 1–6. ACM, New York, NY, USA (2013)
42. Petcu, D., Crăciun, C., Neagul, M., Panica, S., Di Martino, B., Venticinque, S., Rak, M., Aversa, R.: Architecturing a Sky Computing Platform. In: M. Cezon, Y. Wolfsthal (eds.) Proceedings of the International Conference Towards a Service-Based Internet ServiceWave'10, *Lecture Notes in Computer Science*, vol. 6569, pp. 1–13. Springer-Verlag, Berlin, Heidelberg (2011)
43. PingER: Ping end-to-end reporting (2013). URL `http://www-iepm.slac.stanford.edu/pinger/`
44. Ramirez, A.O.: Three-Tier Architecture. Linux Journal **2000**(75) (2000)
45. Red Hat: Drools (2015). URL `http://www.drools.org/`
46. Wang, S., Hsu, C.H., Liang, Z., Sun, Q., Yang, F.: Multi-user web service selection based on multi-QoS prediction. Information Systems Frontiers **16**(1), 143–152 (2013)