MultiPaaS - PaaS on Multiple Clouds

by

Shashank Sahani, Vasudeva Varma

in

2014 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)

Karnataka Bangalore, India

Report No: IIIT/TR/2014/-1

Centre for Search and Information Extraction Lab
International Institute of Information Technology
Hyderabad - 500 032, INDIA
October 2014
Multipaas - PaaS on Multiple Clouds

Shashank Sahni
Computer Engineering
International Institute of Information Technology
Hyderabad, AP - 500032
Email: shashank.sahni@research.iiit.ac.in

Vasudeva Varma
Search and Information Extraction Lab
International Institute of Information Technology
Hyderabad, AP - 500032
Email: vv@iiit.ac.in

Abstract—Current PaaS solutions use an underlying IaaS provider and restrict themselves to a few geographical regions. They operate in one/two zones only and use others for failover or disaster recovery. In this paper, we present the design of MultiPaaS, a PaaS solution which runs on multiple cloud/IaaS providers and leverages the combined global span. Unlike traditional scaling techniques where app servers are scaled up or down in a specific region or data center, we present a design to perform it on a global scale. Origin of the request burst is identified and instances are booted in a data center closest to it. It ensures database persistence and cross-region scalability via a combination of replication techniques. We identify common features available across all IaaS providers and use that as a provider interface to ensure interoperability, but use provider specific extensions to leverage unique offerings and benefits.

I. INTRODUCTION

Cloud providers have simplified operations of developers and system administrators by enabling flexible resource provisioning and programmatic access. Users pay for what they use, making experimentation and deployment cheaper. This has resulted in rapid cloud adoption by small companies and startups who can afford costs incurred from an on-premise infrastructure.

The cloud computing ecosystem comprises of multiple service offerings in the form of a Service model. One of them, Platform as a Service(PaaS), has gained immense popularity among developers for quick deployment of web services. For a single request of developer, PaaS handles system provisioning, base configuration and application deployment. This enables a developer to not waste time or be concerned about configuration, deployment or scaling application. Its automatically and optimally handled by PaaS, making it a popular application hosting solution among developers.

With so much involvement on top of the stack, public PaaS providers prefer offloading infrastructure management by using an Infrastructure as a Service(IaaS) provider. This provides them the benefit of reduced operational overhead of managing physical infrastructure, reduced upfront costs and infinite scaling.

Most of the popular PaaS providers - Docker Inc.(formerly DotCloud)[1], Heroku[2], Pantheon[3], EngineYard[4] etc. operate on present IaaS providers, Amazon Web Services(AWS)[5] being the most popular.

Popular PaaS platforms are primarily used for hosting web services and applications. These are mainly user applications which either provider a web UI or act as a backend service for a mobile application. Both target best user experience demanding the application to be scalable and efficient.

With hosted applications targeting more and more users, its important to have systems closer to the end users. For PaaS a global footprint is imminent. No single IaaS provider can cover the entire globe efficiently. But since PaaS solutions already leverage a pre-existing IaaS, by combining multiple providers, a PaaS can offer a bigger global span.

Interestingly, except Engine Yard[4], all the PaaS providers listed above have restricted their services to one/two regions in U.S. only. But similar to AWS itself, Engine Yard relies on the options provided by the user for region, db replication and instance selection. To effectively leverage a global footprint, the PaaS should smartly handle scaling and geo-balancing requirements.

This can be very helpful in providing optimal delivery globally. Similar to CDN, apps can be spawned in datacenter closer to the traffic origin. With regions come the benefit of automatic failover handling and disaster recovery. Due to the global coverage, organizations which are restricted to operate inside the country due to government regulations will be able to use the PaaS too.

AWS provides spot instances at very cheap rate with no guarantee whereas DigitalOcean[11] provides cheaper monthly fee for on-demand instances. Major IaaS providers like AWS and Rackspace[6], provide access to a lot of vendor products via their marketplace. Such a hybrid environment brings opportunity to utilize specific features and benefits offered by each vendor.

In this work, we present the design of MultiPaas, a multi cloud Platform as a service solution. It aims to address the primary issues of building and running a PaaS on multiple cloud/IaaS providers spread across globe and leverage the benefits offered by them.

The rest of the paper is structured as follows - In section 2 we talk about similar solutions provided by open source softwares and commercial vendors and discuss their shortcomings. In section 3 we briefly discuss the design of PaaS and the applications which run on them. Section 4 talks about challenges in building a multicloud platform. In section 5 we present the design of MultiPaaS. Section 6 explains how scaling and load balancing are handled by MultiPaaS. Section 7 proposes implementation specifics and experiment setup. Section 8 concludes the paper and showcases the vision for the project.
II. RELATED WORK

The ability to use multiple cloud provider has been a challenge since the advent of commercial cloud providers. Plenty of solutions have been developed to tackle different aspects of multicloud environment.

Multicloud libraries exist in almost every programming language to enable the development of applications which leverage multiple cloud providers. jClouds(JAVA)[12], libcloud(Python)[13], fog(ruby)[14], pkgcloud(NodeJS)[15] are a few popular multicloud libraries.

Companies like RightScale[16] and Scalr[17] facilitate the management of multiple cloud providers from a single interface. OnApp[18] will be releasing their provider marketplace via cloud.net[19], providing both a web UI and API to access multiple providers worldwide.

These offerings showcase the demand and requirement of multicloud solutions. All of them provide accessibility to multiple providers from a single interface, but none provide an end to end solution of deploying and scaling an application across providers globally. The tasks of when and where to deploy and scaling up/down are entirely dependent on the user. Amazon Web Services(AWS) being the leading IaaS provider spans 5 continents via 8 different regions, making load balancing across data center and geographic regions an important aspect for their major customers. AWS Elastic load balancer[21], allows application workload distribution across availability zones(data centers in the same region). Based on the instance resource utilization, ELB is also capable of scaling instances on the fly.

AWS DNS offering, Route53[20], enables geo-load balancing via latency based distribution[29]. The endpoint closer to the requesting client(based on low latency) is sent as resolved IP.

These services although provide geographic load balancing and scaling, the latter is strictly regional to mitigate failovers and only operates on a pre-deployed infrastructure. The task of spawning instances in new locations to reduce latency is to be performed by the administrator. Furthermore, this solution integrates very tightly with AWS offerings and results in vendor lock-in.

Our work architects a self scaling Platform as a Service offering which uses multiple cloud providers to span vast geographical regions. Application scaling is automatically handled by identifying increasing load from a region and booting instances in a data center nearest to the end user.

III. DESIGN OF PAAS

Before we discuss the design of MultiPaas and the challenges involved, it would be beneficial to have a preliminary understanding of how a PaaS works and the popular applications hosted on it. This would help us gain a better understanding of the shortcomings and challenges when the design is modified to work on multiple cloud providers.

A. Working of a PaaS

A generic platform as a service offering has the following primary components

- **Controller** - Controller node handles lifecycle of application on the platform. It exposes an API which is utilized by the developer to operate PaaS.
- **Routing/Load balancers** - Load balancers act as the web endpoint for the incoming requests to applications. The load balancers route the request to the appropriate application based on the hostname used in the request. They are also capable of distributing request load based on the instance health/usage and may even scale if the underlying application servers are exhausted.
- **Application Servers** - Application servers are usually virtual machines or containers which host the clients application. These servers can also be used to host Database required by the application.

B. Application Architecture

Common applications hosted on PaaS are three tier web applications[30] comprising of presentation layer, application/logic layer and data layer. The presentation layer is the web interface available to the user. Logic layer hosts the application code and data layer usually hosts the data store required by the application in logic layer. Presentation layers are usually scaled via Content Delivery Networks which can be easily integrated with a web application. The response time and user experience for the presentation layer are dependent on the performance and efficiency of the underlying layers. Hence, with respect to MultiPaas, we will be focus on scaling application layer and data layer.

IV. CHALLENGES

PaaS running on multiple providers globally provides a great opportunity for scaling. Based on the origin of the incoming requests, application servers can be placed closer to the users, improving the delivery time and performance. This, however, introduces few problems.
1) **When to Scale?** With requests originating from different locations, PaaS should be able to quickly identify when scaling is needed. If it responds in haste to a short term request burst, scaling will go in vain in addition to extra cost incurred. If response is too slow, the performance will take a hit.

OpenShift, a PaaS solution from RedHat, uses number of requests as a measure to decide whether to scale or not. But, with the ability to spawn/scale applications in multiple locations worldwide, performance isn’t the only criteria for deciding whether to scale or not. If a majority of the requests are originating from halfway around the world, one can gain significant benefits by placing the app servers in a data center closer to that region. Identifying a suitable datacenter closer to the location of requests origin is needed.

2) **Where to Scale?** In order to scale geographically, it is very important to correctly identify the origin of the incoming requests.

Cloudflare, a popular hosting solution, leverages anycast for geographically optimized delivery. Anycast, a routing scheme, provides the benefit of routing request to the closest server among a cluster of systems, all listening to the same IP. In CloudFlare, this has been a result of configuring network equipments across their data centers, coordinating with upstream providers etc. But control over hardware isn’t a privilege enjoyed by everyone. PaaS providers leverage underlying IaaS, making Anycast not a viable option.

3) **Route users to the nearest servers/node** - Traditionally, webmasters have been using CDN providers extensively to deliver content to the end users in the most efficient way. This is expensive and is applicable to static content only. In addition to this, a well integrated DNS + Hosting solution can provide the benefit of directing users to the nearest node scaling web apps. CloudFlare has been doing the same for a while.

AWS DNS service Route53 provides a latency based name resolution feature for their services - EC2 instances and Elastic Load balancers. This feature ensures that requesting user is responded with IP address closest to it, based on the latency. This service is tightly integrated with AWS restricting PaaS to one provider only.

4) **Variance with Multiple Providers** Multiple providers introduce multiple services and access interfaces. This causes deployment and architectural changes while trying to provide same service transparently across providers. AWS and Rackspace provide block storage as a service for persistent storage whereas linode[7], digitalocean and other virtual hosting provide persistent storage out of the box. APIs and entities/features of each service differ among providers - IOPs can be controlled in AWS EBS but not in OpenStack. Although there exist plenty of unified API solutions - jCloulds(java), libcloud(Python), fog(Ruby) etc. all of them abstract common features causing exclusive attributes of a cloud provider useless. Such differences need to be considered while designing a provider agnostic solution which is capable of leveraging everyones strength.

5) **Handling data across regions and providers** After identifying the increase in load metrics, most of the PaaS solutions scale by booting a calculated number of nodes of a specific configuration in the same region. Since the service has been operating in the same region, the data resides in the vicinity. For Amazon Web Services, even if the apps are booted across multiple availability zones(AZ) in the same region, due to good connectivity between AZs in a region, the latency and performance degradation is very low.

This however isn’t the case with vendor agnostic geo scaling. Data access over WAN can lead to noticeable performance degradation and data corruption if proper measures aren’t in place. With multiple instances of an app running across different providers globally, maintaining a persistent database with reasonable performance is a challenge.

V. DESIGNING MULTIPAAAS

The design is motivated from some of the major PaaS vendors and opensource PaaS solutions Heroku, CloudFoundry[8], OpenShift[9] and Deis[10]. Due to the architectural similarity and flexibility provided by Deis, we decided to implement our design on it.

To avoid complexity in the architecture, we’ve designed multicloud PaaS as a set of multiple regional PaaS deployments. Every regional PaaS deployment runs a common set of services in their respective regions and export API for management, accessible over region specific hostnames. The design is analogous to how IaaS providers operate - AWS, Rackspace etc.

Note that, since MultiPaaS can be deployed across different IaaS providers, region isn’t specific to any one of them. The region can be granular upto city level.

Components of the Architecture/Design Following services comprise a regional PaaS.

- **Routing Layer** - This layer consists of servers responsible for routing requests smartly to appropriate containers running the concerned application code.
- **Runtime Layer** - This layer consists of servers hosting the containers which host and run clients applications. The layer also hosts data specific containers which host database required by the application. The database isn’t shared among applications and one container hosts only one instance of it.

- **API** - Each regional PaaS installation has an API endpoint. This endpoint is exposed by the controller service of the PaaS and is accessible from a region specific endpoint. This enables a client to start/scale/stop an app in a region via an API call to the correct endpoint.

- **Log Server** - Incoming requests for all servers in the runtime layer are logged centrally at this server.

- **Worker(s)** - This node(s) is responsible for processing log data periodically to update the app, request-frequency per region mapping.

- **Scheduler** - This server is responsible for taking load balancing and scaling decisions. Once a decision is made, it sends scaling request to the API endpoint of the region.

- **Applications** - Apps or specifically web apps are client code which would be hosted on the PaaS. Each virtual machine in the runtime layer would be hosting one application only. The routing layer would be responsible for directing requests to appropriate virtual servers.

### VI. SCALING AND ROUTING

#### A. When to Scale?

OpenShift uses number of requests as the criteria for scaling. If the number of requests for all the nodes reaches 90% of this threshold, new nodes are spawned. One can use the same mechanism for region based requests. For each application if the average number of incoming requests from a region crosses 16/second/server, then boot an instance in that region.

This approach handles both region specific and geographical scaling but makes decisions naively. If this sudden surge in traffic lasts only a few mins, a scheduler using this approach would have booted an instance in the concerned region, wasting resources. The decision making can be optimized by considering average number of connection over a period of time.

Similar to system load representation of unix systems by uptime command, MultiPaaS keeps average number of concurrent requests for 1, 5 and 10 mins for every region. If at any point the average value for a different region is more than 16 in all three cases, a new app server/container is booted in that region.

#### B. Where to Scale?

In order to identify the region, the IPs of the incoming request need to be mapped to a geographic region. We are using Maxminds GeoIP database/API[24] to accomplish this. Each query for an IP returns the country, region and latitude/longitude of the origin. We use the geospatial distance to find the nearest data center to the origin of the request.

This requires us to manually maintain geospatial location of every datacenter/region for each provider. Since every provider has few regions each covering multiple countries, the total is handful, making this data manually maintainable.

The worker nodes parse the web request logs collected from the load balancers to calculate request frequency per region per app. Since the typical http log format, fetch the geo location for the client IP. The result is first looked up in a cache datastore. In case of a miss, its requested using the GeoIP API. Based on the location fetched, find the closest service provider geographically. Update the app, provider-frequency mapping for that minute. At the end, the process calculates the frequency of the requests per region per app for over a period of 1, 5 and 10 minute.

The scheduler will loop over this processed data and make scaling decisions. If the threshold is reached for the same region as the regional PaaS itself, scale horizontally in the same region.

If the threshold is reached for a different region, make an appropriate call to the API server to spawn app instances in a datacenter of that region.

#### C. Route users to nearest node/server

Major CDN providers like CloudFlare[28] use anycast to optimize content delivery by sending requests to the closest node. Anycast routes the request to the closest server among a cluster of systems, all listening to the same IP. This requires control over networking equipments and involvement with upstream ISPs which isn’t possible for most of PaaS providers.

We solve this issue through DNS. Initial DNS protocol only shows the IP of the requesting dns client which is mostly ISPs DNS server for recursive name resolution. The new edns-client[31] support added to DNS protocol allows DNS server access to the requesting clients IP instead of only ISPs DNS server IP. This enabled the server to identify the location of the end user and resolve the hostname to the geographically closest server.

IPs of the new nodes - result of scaling, will be added to the DNS server A record by the scheduler. A new DNS request from the end user would automatically direct the request to the nearest nodes.

#### D. Variance with Multiple Providers

We are building a vendor agnostic PaaS that scales globally. In order to ensure biggest global footprint, integration with multiple IaaS providers is required. With multiple providers come differences in service offerings, API, price, performance etc. In order to ensure seamless integration and transparency to customers, MultiPaaS will use minimal services which are common across most of the providers - compute and block storage. Compute nodes will be used to host the application and block storage to host the persistent database.

Note that, similar to other PaaS environments no provider specific credentials are required from the user. All provider accounts are specific to MultiPaaS only.
E. Data Persistence Globally

Popular providers like Heroku provide database for the hosted web applications. With multiple instances of the web service running across different providers globally, maintaining a persistent database with appropriate performance is a challenge.

In order to provide geographical access along with data persistence, database will be deployed in master-master or master-slave clusters. The cluster will scaled along with the application.

VII. Prototype Proposal and Evaluation

We propose to implement the prototype by extending Deis[10], free and opensource PaaS offering supported by OpDemand. The platform is written in Python and uses docker containers to host applications. It is capable of supporting various hosting platforms via pluggable modules making it vendor agnostic.

For the geo capable DNS, we will be using gslb[23]. It also uses maxminds GeoIP database, ensuring that the nearest provider would be same for both MultiPaaS and gslb.

The geographic location data of the cloud providers data centers would be stored in MongoDB database. The geospatial function, geonear[25], provided by Mongodb would allow us to query the nearest datacenter for each requesting IP quickly.

For the prototype, we will be restricting database to Perconas MySQL[26]. The database will be hosted as a master-master cluster on the block storage volumes and will be accessible over network to the application instances. A cluster would ensure easy scalability of the database across regions.

A. Evaluation

For MultiPaaS evaluation, we will host a sample web application on it which exposes only REST API . The API would provide a mix of read and write queries. This would enable the performance testing of the application based on the response time for different requests.

For the setup, we will be using AWS and Rackspace. These two are the leading cloud providers and together provide a wider global span. As per our design, MultiPaaS will be running on each region of these providers as multiple independent PaaS instance. The sample application will be uploaded to one of the regional PaaS and started.

A sample of servers will be booted in each region running a test script which would send requests to the applications API. This will help simulate global access. MultiPaaS will scale applications globally based on the request burst. The test script would record the response time and server for each API request.

The experiment would help us gain the following insights.

- Load incurred by the systems during the pre-scaling phase.
- Time taken to scale an application in a different geographical region.
- Time taken for the application load to settle down after scaling.
- A better estimate of the number of requests/app/region which should be used to make scaling decisions.

VIII. Conclusion and Future Work

We present a design of MultiPaaS, a Platform as a service offering which is capable of running on multiple cloud providers. We discuss the details of scaling an application across multiple providers enabling geographic load balancing. We further discuss implementation specifics on Deis, an open-source PaaS written in Python.

Our next target is to implement the solution on Deis and perform tests to evaluate the platform. This would help us gain insights into the practical bottlenecks incurred by the current design, helping us further improve it.

REFERENCES