Prototyping a High Availability PaaS: Performance Analysis and Lessons Learned

Marcos Machado, Daniel Rosendo,
Demis Gomes, André Moreira,
Moisés Bezerra, Djamel Sadok
Federal University of Pernambuco
Recife, Brazil
Email: {marcos.machado, daniel.rosendo,
demis.gomes, andre.moreira, moises,
jamel}@gprt.ufpe.br

Patricia Takako Endo
University of Pernambuco
Caruaru, Brazil
Email: patricia.endo@upe.br

Calin Curescu
Ericsson Research
Kista, Sweden
Email: calin.curescu@ericsson.com

***Abstract*—With cloud computing consolidation, Platform-as-a-Service (PaaS) has been used as a solution for developing applications with low cost and maximum flexibility. However, an open challenge related to PaaS is the proper handling of multi-tier and stateful applications with support for high availability (HA); and scalability can be considered an essential feature for HA. However, dealing with several instances of the same application that access its state in a common area is not a simple task. This paper presents a novel PaaS framework, named NoPaaS, that supports the deployment of multi-tier and stateful applications assuring their availability according to the Service Availability Forum (SAF) redundancy model. The primary goal of this work is to present NoPaaS framework and prototype, and highlight challenges and open issues when providing multi-tier and stateful applications in high availability clouds.**

I. Introduction

Cloud services have been used for many purposes by diverse types of commercial enterprises and governmental entities around the world. Looking from a client perspective, cloud computing is very attractive because it minimizes infrastructure investments, resource management costs, and at the same time, presents a flexible business plan (pay-as-you-go pricing) with an elastic service provisioning, in which services can be scaled up or down on demand. On the other hand, from a cloud provider point of view, managing the cloud infrastructure remains a great challenge since all clients requirements should be attended with next to zero outages ([1], [2]).

A report [3] from the International Working Group on Cloud Computing Resiliency (IWGCR) presents a brief summary of availability of major cloud providers. Across different businesses, the downtime cost has a great economic impact for providers. For instance, an airline reservation operation incurs a loss of around $2,600,000 per hour; and a brokerage operation has a cost of $6,450,000 per hour. In order to mitigate the outages, Cloud providers have been focusing on ways to improve their infrastructures and management strategies to achieve high availability (HA) services.

In [4], availability is defined for a given interval of time in terms of the percentage of time the application is up and its services are provided; high availability is achieved when the outage is less than 5.25 minutes per year [4], meaning at least 99.999% availability (five nines). According to [4], HA systems are fault tolerant systems with no single point of failure; in other words, when a system component fails, it should not imply the outage of the service provided by that component.

However, achieving and maintaining high availability is not a trivial task; this is because we are dealing with cloud applications, which usually are composed of several and dependent tiers (multi-tier and stateful applications). Each tier is responsible for a part of the whole system, and they work together to provide a complete service. Although grouped as a whole, in practice each tier is an independent resource for the cloud provider (a virtual machine or a container, for instance) and may be treated as such. This scenario brings many challenges and questions like ”*how should we handle the scaling of multi-tier applications?*” and ”*how should we deal with load balancing among tiers?*” These are still open issues.

The main contributions of this work are: a) propose and describe an architecture for high availability clouds with support for multi-tier and stateful applications deployment, named NoPaaS (Novel PaaS); and b) describe the prototype development and evaluation, highlighting some lessons learned during the process.

II. NoPaaS Framework

In order to define our framework for high availability clouds, we grouped requirements into two categories: a) application requirements, and b) framework requirements. Application requirements represent mandatory characteristics for all applications to work properly within the NoPaaS framework (Table I). On the other hand, framework requirements are a set of services and characteristics that the NoPaaS framework itself must provide to applications and/or developers (Table II).

Considering all these requirements, we propose our framework for high availability clouds, named **NoPaaS** (Novel PaaS), as shown in Figure 1. The NoPaaS was designed to support multi-tier and stateful application deployment, and for

TABLE I
APPLICATION REQUIREMENTS

REQ A.1	Must implement the API as described by the framework
REQ A.2	Must always include the session ID in messages exchanged between tiers
REQ A.3	RESTful communications between tiers

TABLE II
FRAMEWORK REQUIREMENTS

REQ F.1	Must define a REST API for communication with applications
REQ F.2	Must support different profiles configurations
REQ F.3	Must plan resource allocation based on different profiles configuration
REQ. F.4	Must support multi-tier applications
REQ. F.5	Must support stateful applications
REQ. F.6	Must deal with sticky sessions
REQ. F.7	Must assure HA
REQ. F.8	Must provide scaling
REQ. F.9	Must provide resource management
REQ. F.10	Must rely on Cloud infrastructure compatible with current standards

that, provides services for high availability, such as checkpoint, session migration, and failure recovery.

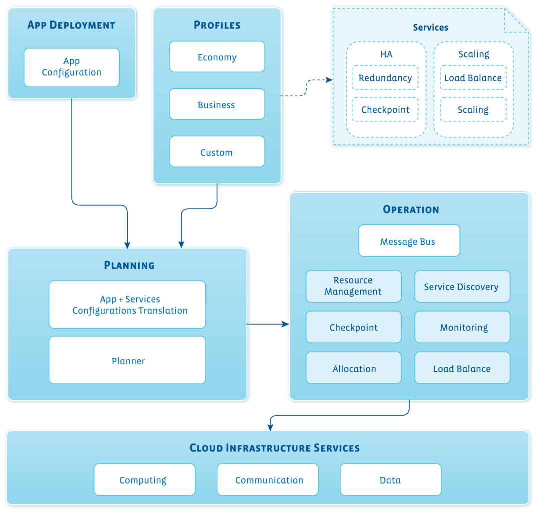

Fig. 1. NoPaaS Framework

1) App Deployment module: it is responsible for the interface between our framework and the developer (an actor willing to deploy a service in our framework). To avoid the need to "reinvent the wheel", NoPaaS proposes a set of modules that should work alongside a PaaS service, extending it. Such modules must act as a gateway between the PaaS service and NoPaaS internal services. Services which will be deployed within our framework must accomplish **REQs A.1, A.2, and A.3** regarding the application, and **REQ F.1** regarding framework requirements.

2) Profiles module: Profiles are a way to represent the available budget and requirements into response time and availability levels. The NoPaaS has, but it is not limited to, three different profiles: a) economy; b) business; and c) custom. For each profile, we define a specific configuration regarding load balancing, scaling, checkpoint mechanism, and redundancy model based on the Service Availability Forum (SAForum or just SAF) model. **REQs F.2 and F.3** are obeyed by this module.

3) Planning module: The set of information provided by the developer regarding application's configurations, and profile are sent to the *App + Services Configurations Translation* in the Planning module, which is responsible for translating these information to be used by the Planner. The Planner evaluates requirements, available resources, and plans the ideal deployment of the application (according to the SAF redundancy models) to satisfy **REQs F.4, F.5, and F.6**. The Planner also maintains direct contact with the Resource Management entity (in the Operation module) for an updated view of resource availability. The Planner is responsible for executing two main activities: estimating the availability based on SAF redundancy models for a given application configuration, and defining the allocation of an application to minimize costs and reach a minimum availability requested by a client. We are considering that each tier of an application is abstracted as a Service Instance (SI), and each SI is assigned into a Service Unit (SU). The Planner uses the analytic worst-case models to estimate availability of each protection configuration. For a detailed explanation about how our analytic models estimate availability and simulation results, please see [5].

4) Operation module: provides services to deal with the cloud infrastructure and application resources. The Resource Management is responsible for supervising the infrastructure, alerting upon application failures and generating scaling in/out triggers. The Checkpoint stores backups of deployed applications, recovering their states in case of failure, and also deals with session migrations. The Allocation enforces the reservation of resources designed by the Planner. The Monitoring keeps track of applications and physical resources, maintaining a map of resource usage. The Load Balance is used to distribute the load among multiple tiers of an application, dealing with session stickiness, server failure, and session migration. We define the Message Bus entity for communication purposes, and it is responsible for receiving and delivering messages for all entities. **REQs F.7, F.8, and F.9** should be attended by services of this module.

5) Cloud Infrastructure module: The Cloud Infrastructure services comprise the IaaS services that NoPaaS will use to deploy and manage developers' applications. The main idea is to use Cloud facilities in order to avoid unnecessary work. For that, we need to hire an IaaS provider or configure our private IaaS. With this, we comply with **REQ F.10**.

III. Prototype Description

Figure 2 shows how our prototype implementation reflects the NoPaaS architecture proposed in Section II. In the prototype, some modules and services were renamed in order to explain their respective functions.

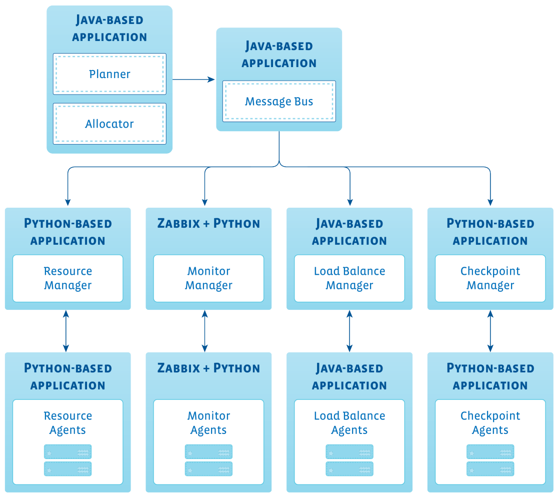

Fig. 2. NoPaaS Prototype

A. Software infrastructure

Our software platform is based on Juju and Openstack. Juju uses Openstack to manage and deploy VM instances, creating VMs in which the framework can deploy its applications in a simple drag-and-drop fashion. The Operating System used was Ubuntu 15.04, the version of Juju was 1.25.0-vivid-amd64, and the version of the Openstack was 1.0.3.

1) Planner: handles the interactions with Juju to create the VMs needed to deploy applications, managers, and agents, allocating all the resources required. We chose to use the 2N redundancy model statically configured to deploy applications for the sake of our evaluations and tests. Furthermore, the Planner also is responsible for the new entities allocation when we need to scale up or down. Lastly, it handles the reassignment of failed instances.

2) Allocator: is the entity that enforces the allocation plan given by the Planner. When the developer deploys a new application, the Planner request the Allocator to create the necessary SIs inside the available SUs. The Allocator has a central unit that runs alongside the Planner and receives the information to perform the allocation of new instances; it also has a remote unit that runs inside of each SU. The remote unit is responsible for the instances' deployment.

3) Resource Manager: uses its agents to supervise all SUs and their SIs individually. It receives monitoring alerts from the Monitor Manager and validates them, confirming the status with the applications and the used resources with its agents. The incoming alerts may cause the Resource manager to reallocate resources due to a scaling down or scaling up requests. Moreover, reassignments may be done because of applications or SI failures. To operate that, it communicates with the Allocator.

4) Load Balance: manager and agents work together to deal with sticky sessions, server failure, and session migration. The manager is a centralized entity with a complete view of all agents spread in different tiers. It is responsible for distributing stats, session information, and message routing information to all agents. The Load Balance agents were divided into border and internal agents. Border agents are openly accessible and are responsible for forwarding users' requests from the front tier. Internal agents are located in each SI and only receive requests from the application, so it is not directly accessed by external clients.

5) Monitor: was implemented using Zabbix version 2.4 and its Python API to automate the control of the Monitor manager over the Zabbix Server. Zabbix uses an agent-proxy-server architecture and can oversee a variety of metrics. The manager keeps track of the metrics status, updating the Load Balance manager and the Resource manager, sending alerts upon failure detection, or an underused or overused resource.

6) Checkpoint: manager stores application's state, recovers state in a standby replica and deals with session migration. The Checkpoint agents are executed in the same SU of an application; they collect information via REST and send them to the Checkpoint manager.

B. Hardware infrastructure

We deployed our prototype into an Intel Xeon CPU E5410, with 40 Gb of RAM, and two hard drives; one is a Western Digital WD5003ABYX-1 with 500 Gb and one is a Seagate ST2000DM001-1ER1 with 2 Tb.

IV. Evaluation and Experimental Results

As a proof of concept, we developed a chat application composed of three tiers: front-end, back-end, and database.

A. Deployment Time

The deployment time is the total time needed to deploy the multi-tier application from a developer request until the application is formally accessible for clients, i.e., all tiers and services are ready. For this metric, we have two different measurements, named environment deployment (the time elapsed from the beginning of the deployment until the end of the NoPaaS environment setup. This metric is related to NoPaaS mechanisms to deploy an application) and application deployment (the time interval from the end of the environment setup until the end of the application deployment. This metric is related to the application configuration process). We made this separation because both periods are independent from each other.

Figure 4 shows the boxplot of deployment duration measured in our NoPaaS prototype. As one can note, the environment deployment is faster and presents fewer oscillations than the application deployment. Table III shows that our environment deployment average time is less than half of the application deployment average, presenting around 20 seconds, in the worst case.

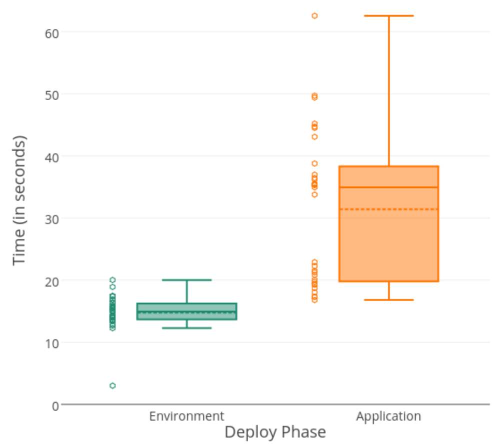

Fig. 3. Deployment Time

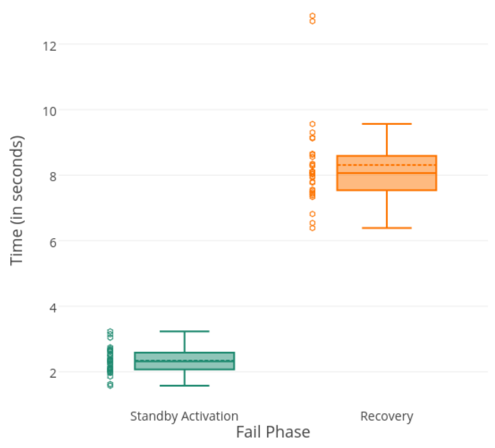

Fig. 4. Failure Recovery Time

TABLE III
DEPLOYMENT TIME

Metrics	Deployment time (seconds)		
	Environment	Application	Total
Average	14.77252	31.39755	46.17008
Median	14.95152	34.94713	49.04846
Confidence Interval (95%)	0.99349	4.30932	3.82115
Maximum Value	20.03021	62.55691	65.56328
Minimum Value	3.00637	16.80035	31.46902

TABLE IV
FAILURE RECOVERY TIME

Metrics	Failure Recovery Time (seconds)	
	Standby to Active	Recovery
Average	2.33777	8.30916
Median	2.32032	8.06126
Confidence Interval (95%)	0.14	0.48
Maximun Value	3.23023	12.86280
Minimum Value	1.57473	6.38666

These results show that the NoPaaS prototype has a small impact on the total time needed to deploy a multi-tier application, highlighting that the developers need to be concerned about and are free to decide how their applications will be implemented regarding the services of each tier. Furthermore, the application deployment time had some oscillations that we assign to the multi-tier application deployment, in which each part of the application in each tier must be online, with the communication established between each tier, and the back-end tier's application must connect to the database in the database tier. Only after all these steps does the application report to the monitor, which reports to the NoPaaS framework.

B. *Failure Recovery Time*

Each SI can be assigned as active or standby. When a SI fails, the framework a) activates the standby SI and migrates sessions; and b)tries to recover the failed SI and changes it states to standby. Figure 4 shows the required time (in seconds) to fulfill these two tasks, respectively.

The first task (modify the SI instance state from standby to active and migrate sessions) requires an average of 2.33 seconds, while the second task requires about 8.3 seconds (see Table IV). The second task requires more time than the first one, because this time refers to the whole recovery process (from the receiving of the failure alert until the allocation of the failed machine).

V. CONCLUSIONS AND FUTURE WORKS

In this paper, we presented the NoPaaS, a framework focused on providing high available services with support to multi-tier and stateful applications. Our experimental results showed good performance regarding deployment and failure recovery times. As future works, we plan to evaluate other high availability services, such as checkpoint and load balance, and compare NoPaaS with other existing frameworks.

VI. ACKNOWLEDGMENTS

This work was supported by the RLAM Innovation Center, Ericsson Telecomunicações S.A., Brazil.

REFERENCES

[1] D. Puthal, B. Sahoo, S. Mishra, and S. Swain, "Cloud computing features, issues, and challenges: a big picture," in *Computational Intelligence and Networks (CINE), 2015 International Conference on*, pp. 116–123, IEEE, 2015.
[2] N. L. da Fonseca and R. Boutaba, "Cloud architectures, networks, services, and management," 2015.
[3] C. Cérin, C. Coti, P. Delort, F. Diaz, M. Gagnaire, Q. Gaumer, N. Guillaume, J. Lous, S. Lubiarz, J. Raffaelli, *et al.*, "Downtime statistics of current cloud solutions," *International Working Group on Cloud Computing Resiliency, Tech. Rep*, 2013.
[4] M. Toeroe and F. Tam, *Service availability: principles and practice*. John Wiley & Sons, 2012.
[5] G. Gonçalves, P. Endo, M. Rodrigues, D. Sadok, and C. Curescu, "Risk-based model for availability estimation of saf redundancy models,"