DisOpenFaaS: A Distributed Function-as-a-Service Platform

Lucas Vieira[1], **Adbys Vasconcelos**[1], **Ítalo Batista**[1], **Rodolfo Silva**[1], **Francisco Brasileiro**[1]

[1]Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Departamento de Sistemas e Computação
Laboratório de Sistemas Distribuídos
Av. Aprígio Veloso, s/n, Bloco CO
58.429-900, Campina Grande, PB – Brazil

{lucas.vieira, adbys, italobatista, rodolfoams, fubica}@lsd.ufcg.edu.br

Abstract. *The widespread adoption of cloud computing leads to a constant development of more efficient ways to manage the resources used by it. One of the most recent steps is called Function-as-a-Service (FaaS), a cloud computing service model where developers can focus on developing functions, without worrying with the underlying resources. We propose an architecture for deploying FaaS platforms in hybrid clouds that can be composed by multiple cloud providers. The architecture enables privately deployed FaaS platforms to perform auto-scaling of virtual machines in a distributed infrastructure and routes requests to servers geographically located closest to the clients, considering a scenario where the users of such platform are scattered around the globe.*

1. Introduction

During the last decade, more and more companies and individuals have adopted cloud computing as a means of providing Information Technology services. With that increased interest and adoption of cloud computing, multiple deployment models have been proposed for such infrastructures, allowing it to be used not only via big cloud providers, namely *public clouds*, as they were in the beginning, but also via *private clouds*, using companies' own infrastructure. Lately, there has also emerged a demand for *hybrid cloud* deployments, *i.e.*, a deployment model that uses both public clouds and private clouds, integrating them into a single environment, namely multi-cloud infrastructures. According to RightScale[1], 81% of companies that rely on cloud computing to provide their services choose to use a multi-cloud approach [RightScale 2018].

Another aspect that has changed a lot in the cloud computing scenario is the number of service models that are available. Classic service models were restricted to SaaS – Software as a Service, PaaS – Platform as a Service, and IaaS – Infrastructure as a Service, with those models differing on the level of control and management responsibilities that the users had over the cloud infrastructure. In contrast, almost anything is provided as a service and offered by cloud computing providers recently. In this work we will be highlighting one of such service that has gained a lot of popularity among cloud computing researchers and developers, FaaS – Function as a Service.

The FaaS service model aims at shortening an application development cycle by allowing developers to focus on the development of highly-specialized code (**functions**)

[1]https://www.rightscale.com/

that performs a single task, and deploy them in highly scalable and highly available infrastructures, without having to dealdirectly with the underlying physical and virtual infrastructure where they will be hosted and executed. In such environments it is much easier and faster to test the newly created functions. Besides that, a much more fine-grained, pay-per-use billing model can be used, considering the number of requests to use a function, and the processing power used by it.

Early FaaS services date back to 2014, when the big cloud providers started to make them available[2], but in the last couple of years, several FaaS platforms have emerged [OpenFaaS , OpenWhisk , Kubeless , Fission], allowing the FaaS service model to be deployed also in private or hybrid cloud computing environments.

A FaaS platform usually consists of a central component – API Gateway – which is responsible for receiving requests to deploy, update, remove or trigger the execution of functions, and a backend platform that can be use to automatically scale the resources allocated to a function depending on the demand (and possibly some other user-defined limits). It is important to note that most of the FaaS platforms that are available for private deployments, can only make auto-scaling of containers where the functions are executed, but not auto-scaling of nodes (virtual machines or bare-metal nodes).

Considering the scenario of multi-cloud platforms, as well as the possibility that its users are spread worldwide, it would be wise to have requests made by users be handled by services hosted as close as possible to them.

In this paper we propose a tool forimplementing distributed platforms that support FaaS, considering both multi-cloud and federated cloud scenarios, that can effectively balance the load between multiple instances of the service, as well as auto-scale computational infrastructure capacity available for executing each registered function, considering both virtual machines and containers.

The rest of this work is divided as follows: in Section 2 we describe the a architecture proposed for deploying FaaS-based services in geographically distributed multi-clouds. Thereafter, in Section 3 we discuss the tool hereby proposed, which is a reference implementation of the proposed architecture. This is followed by the description of the documentation of the proposed tool (Section 4), and how we intend to demonstrate the tool at the event (Section 5). We conclude the paper with our final remarks and directions for future work.

2. Architecture

In this section we describe the architecture proposed for deploying and executing Serverless applications in federated clouds. This architecture comprises three main components: (i) **FaaS Cluster**, (ii) **FaaS Proxy**, and (iii) **Multi-Cloud Resource Allocator** (MCRA). Such architecture is depicted in Figure 1 [Vasconcelos et al. 2019].

The **FaaS Cluster**, is a cluster where the actual FaaS platform is deployed. Multiple instances of the FaaS Cluster are deployed on geographically distributed cloud infrastructures. This component receives and processes requests coming from the FaaS Proxy. It is also responsible for monitoring the resources used in each individual cloud and request the MCRA to perform the auto-scaling of nodes.

[2] `https://aws.amazon.com/lambda/`

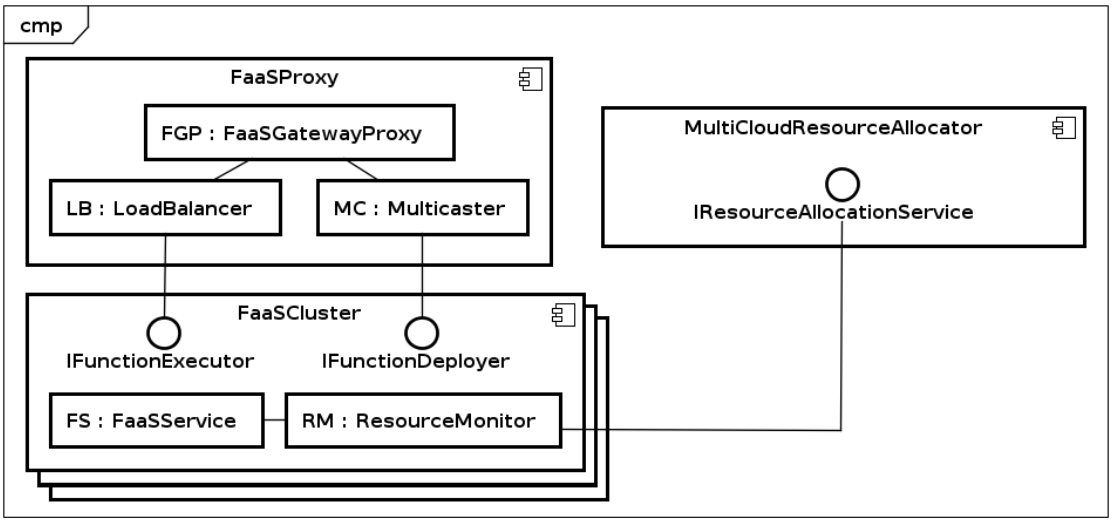

Figure 1. DistributedFaaS architecture.

The **FaaS Proxy** is the new entry-point of the entire architecture. It provides the same interface that a regular FaaS gateway would. However, instead of processing the requests directly, it chooses between either forwarding the request to a load balancing service or forwarding it to a multicasting service. This choice depends on the type of the request received. Requests related to registering or removing functions available should be forwarded to a multicaster service. This service will make sure the received requests are replicated in all FaaS Clusters deployed. On the other hand, requests for executing functions are forwarded to a load balancing service. This service will chose where to execute the function, based on the geographic location of the user who made the request and that of the available FaaS Clusters.

Last but not least, the **Multi-Cloud Resource Allocator** (MCRA) is the component that allocates or removes resources in any of the clouds that provide the infrastructure to the FaaS Clusters. These clouds may be operated by different providers, and run different orchestrator middleware. The MCRA implements auto-scaling at the virtual machine level.

3. Implementation

In this paper we provide a reference implementation of the architecture proposed based on some of the most prominent solutions in cloud computing at this time: Fogbow[3], OpenFaaS[4], Kubernetes[5], HAProxy[6] and Asperathos[7]. Fogbow is used for the provisioning of virtual machines in a heterogeneous multi-cloud setting, where the FaaS Clusters will be deployed, and also to allow the auto-scaling done by the MCRA. OpenFaaS was chosen as the FaaS platform of the individual FaaS Clusters. It is used for deploying and executing functions, as well as performing intra-cluster auto-scaling of resources (contain-

[3] http://www.fogbowcloud.org/
[4] https://www.openfaas.com/
[5] https://kubernetes.io/
[6] http://www.haproxy.org/
[7] https://github.com/ufcg-lsd/asperathos

ers). Kubernetes is the backend technology used by OpenFaaS to deploy its infrastrucure. HAProxy, on its turn, is used to implement load balancing at the FaaS Proxy component. Finally, Asperathos is used for monitoring the usage of physical resources (memory, CPU, etc.) at each FaaS Cluster, and perform the auto-scaling of virtual machines using Fogbow. The implementation of the FaaS Proxy is complemented with a simple multicaster explained in Section 3.1. In the rest of this section we further describe the implementation of each component, as well as the communication between them.

3.1. FaaS Proxy

The FaaS proxy is composed by three subcomponents: (i) an OpenFaaS Gateway Proxy, a (ii) Load Balancer, and a (iii) Multicaster.

The OpenFaaS Gateway Proxy is an HTTP(S) server written in Golang that exposes the same API as a regular OpenFaaS API Gateway. That is, the server is able to take requests to create, update, and execute functions, as well as requests for deleting functions that are no longer needed. When a request for creating, updating or deleting a function is received, the OpenFaaS Gateway Proxy forwards it to the FaaS Proxy Multicaster, so that the message can then be sent to all OpenFaaS Clusters belonging to the same cloud federation. On the other, if a request to execute a function is received, it is forwarded to the FaaS Proxy Load Balancer, so that it can choose to which cloud provider should the request be forwarded, based on the location of the user that performed the request and the location of the OpenFaas Clusters.

The FaaS Proxy Multicaster, also developed in Golang, receives requests from the FaaS Gateway Proxy and replicates such requests across all FaaS Clusters currently deployed. If a request fails to be executed in any of the FaaS Clusters, the Multicaster is responsible for retrying it in background, until it succeeds. The Multicaster also keeps a history of all actions performed in time, so that if a new FaaS Cluster joins the cloud federation, it can replicate the state of the other FaaS Clusters (deploying the currently available functions).

The FaaS Proxy Load Balancer, implemented using HAProxy, uses a set of rules configured as Access Control Lists (ACLs) and the information about the client geographic location (based on the client's IP address) to determine to which FaaS Cluster is closest to the client and thus should process the request.

3.2. FaaS Cluster

In our imlpementation, the FaaS Cluster is based on the OpenFaaS platform. It can be subdivided into two main subcomponents: (i) FaaS Service, and (ii) Resource Monitor, further described as follows.

The FaaS Service is an instance of the OpenFaaS platform. OpenFaaS can use many backend technologies, such as Docker Swarm and Kubernetes. We have chosen to use Kubernetes as the underlying technology for deploying it, as it eases the task of auto-deploying and auto-scaling the containers that compose a given application or service. The OpenFaaS platform consists of an API Gateway, responsible for receiving requests to create, update execute and remove functions, as well as containers of the functions themselves. Additionally, we have included a deployment of a Kubernetes Metrics API[8],

[8] `https://kubernetes.io/docs/tasks/debug-application-cluster/`

that enables us to collect metrics about the resources (CPU, memory, etc.) used by the OpenFaaS cluster.

The FaaS Cluster Resource Monitor was implemented in Python using Asperathos, which is a platform to facilitate the deployment and control of applications running in cloud environments. For the purposes of this work, we have developed three *Asperathos plugins* that are responsible for: (i) collecting metrics from the Kubernetes Metrics API, (ii) analysing the need for new resources (virtual machines) or releasing resources that are no longer needed, and (iii) making the actual requests to the MCRA to do the auto-scaling of nodes.

3.3. Multi-Cloud Resource Allocator

The MCRA, based on the Fogbow middleware, is responsible for receiving requests from the FaaS Cluster Resource Monitor to allocate or release resources. Upon receiving a request, the MCRA uses the Fogbow Resource Allocation Service (RAS), and asks it to make the necessary adjustments (adding or removing virtual machines) to the appropriate cloud where the request came from. This component, together with the FaaS Cluster enables our solution to achieve the aimed auto-scaling of nodes.

3.4. Components Interaction

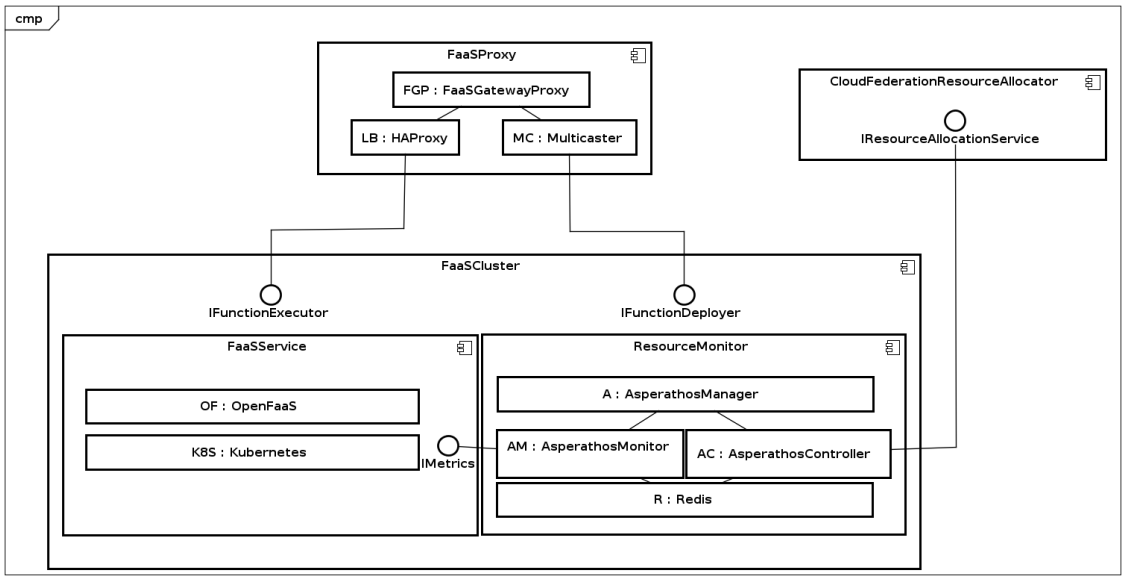

Figure 2. DistributedFaaS communication architecture.

The communication among the components used in our tool is shown in Figure 2 and is better detailed in the next sections.

3.4.1. FaaS Proxy Components Interaction

The OpenFaaS Gateway Proxy is the system entry point. It receives HTTP(S) requests, analyzes and forwards them for the next component. Requests for creating, updating

resource-usage-monitoring/#resource-metrics-pipeline

and deleting functions are forwarded, without modifications, to the Multicaster. Requests to invoke a function are forwarded to the Loadbalancer Component but it has an *X-Forwarded-For* header added to it in order to maintain the original request source IP.

Once the Multicaster receives an HTTP(S) request, it will replicate such request over all FaaS Clusters. The Multicaster will use an HTTP(S) request to communicate with the FaaS Clusters.

The Load Balancer component is an HAProxy server. The Load Balancer will receive an HTTP(S) request and forward it to the closest FaaS Cluster. HAProxy will look into the X-Forwarded-For header to obtain the source IP address and then find the FaaS Cluster closest to the client.

3.4.2. Resource Monitor Components Interaction

An HTTP(S) request is required to start monitoring FaaS Cluster Components. A request containing configuration for Asperathos Components is addressed to Asperathos Manager. Asperathos Manager will receive the request and forward the respective configuration for each Asperathos Component over an HTTP(S) connection. Asperathos Components will start right after receiving the required configurations.

The Asperathos Monitor will query Kubernetes API periodically over an HTTP(S) connection to get FaaS Cluster CPU usage information. As soon as the information is retrieved, Asperathos Monitor will save it on Redis.

The Asperathos Controller will check Redis regularly to get CPU usage information. When new CPU usage information is available and scaling is necessary, the Asperathos Controller will make a Remote Procedure Call to the MCRA (Fogbow Proxy) which will communicate with Fogbow using its Rest API.

4. Documentation and source code

The documentation of the tool has two parts: i) the installation guide, and ii) the API documentation.

The installation guide is available online, and can be accessed in the following URL: `https://git.lsd.ufcg.edu.br/distributed-faas/distributed-faas-demo`.

Since the DisOpenFaaS tool provides the same API as the standard OpenFaaS tool (implemented by its OpenFaaS Gateway Proxy component), users of the DisOpenFaaS tool should consult the OpenFaaS REST API documentation available online, and accessible through the following URL: `https://app.swaggerhub.com/apis/adbys/FaaSProxyGateway/0.0.1`.

Additionally, users can access the videos that demonstrate how the tool is installed (available online at `https://youtu.be/-dLA\_Fbgt80`), and how it works (available online at `https://youtu.be/pGuiV7Oo1hQ`).

4.1. Source code

This project has been funded by a private company, and the code is not public. Nevertheless, for the sole purpose of the evaluation of the tool, we are providing the reviewers

with access to our private software repository. The software repository is available at `https://git.lsd.ufcg.edu.br/distributed-faas`, and reviewers can login using the following credentials:

- username: sbrc2019
- password: sbrc2019!

5. Demonstration

The demonstration planned will use a pre-installed deployment of the tool using two clouds at UFCG. In this deployment we use the tool netem () to emulate a WAN environment, that mimics a real setting. Then the demonstration will follow a flow that resembles what is shown in the video available at https://youtu.be/pGuiV7Oo1hQ. Thus, the only requirement for the execution of the demonstration is the possibility to provide our own laptop (where the client software will run) with an Internet connection (ideally a wired one, but it should not be a problem if only Wi-Fi is available).

6. Conclusions

We have implemented a tool that allows the deployment of a distributed FaaS platform. This was done by implementing a new component (FaaS Gateway Proxy) that acts as an interface between the users of the distributed FaaS platform and the actual FaaS Clusters that are deployed in multiple clouds.

The availability and scalability of functions was achieved by the use of OpenFaaS and Kubernetes clusters. By using such technologies, replicas of the functions are automatically deployed and removed in virtual machines belonging to a same FaaS Cluster. We were also able to achieve auto-scaling of virtual machines in a distributed multi-cloud environment, by using the Asperathos framework in conjunction with the Fogbow middleware. This allowed our system to allocate and release both containers and virtual machines seamlessly.

The third goal of this work, distributing the load across the multiple FaaS Clusters based on the geographic location of users who make the requests was achieved by writing ACLs to an HAProxy server proxy.

As a future work, we intend to better evaluate the performance of FaaS-based applications deployed in such platform, as well as to assess the overall benefits regarding resources utilisation.

ACKNOWLEDGEMENTS

This work was partially funded by CAPES and Dell EMC.

References

Fission. Fast serverless functions for kubernetes. https://github.com/fission/fission. Accessed: 04/16/2019.

Kubeless. Kubernetes native serverless framework. https://github.com/kubeless/kubeless. Accessed: 04/16/2019.

OpenFaaS. Openfaas - serverless functions made simple. https://github.com/openfaas/faas. Accessed: 04/16/2019.

OpenWhisk. Apache openwhisk. https://github.com/apache/incubator-openwhisk. Accessed: 04/16/2019.

RightScale (2018). Rightscale 2018 state of the cloud report. https://assets.rightscale.com/uploads/pdfs/RightScale-2018-State-of-the-Cloud-Report.pdf. Last access 12/03/2019.

Vasconcelos, A., Vieira, L., Batista, I., Silva, R., and Brasileiro, F. (2019). Distributed-faas: Execution of containerized serverless applications in multi-cloud infrastructures. In *CLOSER*.