Evaluating Apache OpenWhisk - FaaS

Sebastián Quevedo
Escuela Politécnica del Litoral, ESPOL
Campus Gustavo Galindo, PO-Box 09-01-5863
Guayaquil, Ecuador
Email: asqueved@espol.edu.ec

Freddy Merchán
Escuela Politécnica del Litoral, ESPOL
Campus Gustavo Galindo, PO-Box 09-01-5863
Guayaquil, Ecuador
Email: fjmercha@espol.edu.ec

Rafael Rivadeneira
Escuela Politécnica del Litoral, ESPOL
Campus Gustavo Galindo, PO-Box 09-01-5863
Guayaquil, Ecuador
Email: rrivaden@espol.edu.ec

Federico X.Dominguez
Escuela Politécnica del Litoral, ESPOL
Campus Gustavo Galindo, PO-Box 09-01-5863
Guayaquil, Ecuador
Email: fexadomi@espol.edu.ec

***Abstract*—Function-as-a-Service (FaaS) platforms enable users to execute user-defined functions without worrying about operational issues such as the management of infrastructure resources. In order to improve performance, different FaaS platforms are implementing optimizations and improvements, but it's not clear how good these implementations are.**

In this work, Apache OpenWhisk platform is evaluated from an approach that allows to determinate and characterize the performance under different configuration options; it was found that under certain premises an improvement of the performance in cold-booting latencies up to 38% is obtain.

I. Introduction

Serverless computing has high expectations of use due to the advantages of its implementation. In nowadays, the tendency to run quite small applications on heavier and robust (monolithic) systems is remarkable. The industry around serverless architectures has increased [1, 2, 3, 4] and clearly shows a signal that the trend will continue beyond conventional applications [5, 6].

Serverless computing is a relatively new emerging cloud architecture model, which allows developers to focus on programming their applications with not worrying about infrastructure, allowing a high level of scalability [6].

Function-as-a-Service (FaaS) is a subclass of serverless platforms, where a section of stateless code is executed in response to an event. Provide a low-cost application is something attractiveness of this paradigm, a flexibility where an application consists of individual functions that can be managed and executed separately. In general, the use of this platform is very economical since there is not cost for downtime and just a fraction cost of the used time [7, 8, 9, 10].

FaaS among its benefits, on the one hand allows software developers to dedicate themselves to the logic of their application, forgetting about management's resources knowing the problematic of their administration, and on the other hand cloud providers improve the efficiency of their infrastructure resources. In this way, applications can be deployed and scaled fast without the need of starting new servers [7, 11, 12].

Due as mention above, FaaS model has a promising future and a place in the cloud, however this model brings with it challenges related to performance that may put its implementation in risk. About performance, the low latency that a user expects when generating an event is a problem in the FaaS model. [7] Shows the following observations:

1) Most implementations run each function in separate containers, this leads to start containers in "cold" and finish the container once its execution is done. This causes a long start latency for each request.
2) By keeping a container 'hot' for a while to handle future requests, it reduces the latency of a function call, which involves a unnecessarily cost of occupying system resources during the period of inactivity (ie, inefficiency of the resources).

In order to improve performance, current platforms such as OpenWhisk [13] are implementing optimizations and improvements, but it's not clear how good these implementations are. To deal with this, the evaluation of the Apache OpenWhisk platform is proposed, from an approach that allows to determine and characterize the performance under different configuration options. From this approach, our contribution consists of:

- Determine the default performance of the Apache OpenWhisk platform for function calls.
- Optimize Apache OpenWhisk for better performance in response latencies to a function call.
- How good are the optimizations that are made on the Apache OpenWhisk platform

The rest of the article is organized as follows: section II an Overview of Apache OpenWhisk platform; section III Experiments and platform evaluation and section IV general conclusions of this work.

978-1-7281-3764-3/19/$31.00

II. Apache OpenWhisk Overview

In this section, the server-free computing platform of Apache OpenWhisk is described, its background, deployment, the programming model, and finally the internal processing flow explained from a usage scenario.

A. Background

Apache OpenWhisk is an open source serverless platform that executes functions in response to events at any scale and automatically manages all the infrastructure, services and scaling of applications. OpenWhisk competes against big platforms like Nginx [14], Kafka [15], Docker [16] and CouchDB [17], all these platforms come together to form a serverless cloud service. Finally, this platform offers a Command Line Interface (CLI) called "wsk" to create, execute and manage easily OpenWhisk entities and that can be installed in any operating system, in this way developers can implement and interact with the platform [18].

B. Deployment

Apache OpenWhisk can be deployed and configured on many platforms, because it builds its components using containers, and this allows it to support many deployment options, locally and within a cloud infrastructure. The deployment can be done on platforms such as Kubernetes [19], Mesos [20] and OpenShift [21].

C. Programming Model

Apache OpenWhisck programming model is based on three main points: Action, Trigger and Rules. Where Actions is the stateless function that executes arbitrary code; Trigger is a class of events that come from a variety of sources and Rules allows to assign a trigger to an action. In addition to main points, OpenWhisk allows to set up actions together to form a sequence. The programming model, as an outstanding feature, has the permissiveness of programming functions in various types of programming language such as Java [22], Python [23] and JavaScript [24], among others. OpenWhisk is based on an event-driven architecture where most actions are executed as events occur [25].

D. Processing Internal Flow

To explain the internal mechanisms of the processing flow that Apache OpenWhisk performs, the following scenario is proposed: A software developer wants to create a cloud application that allows the reception of an image, reduce the size of it and save all the data. For this, the developer chooses to use Apache OpenWhisk with JavaScript programming language. The flow process is as follows:

1) The developer make a function and uses the CLI of the platform to send it and create it.
2) The developer invokes the function through an HTTP call.
3) The system entry is through NGINEX which is mainly used as a reverse proxy for the API that forwards the appropriate HTTP calls to the next component. All the requests that arrive to the OpenWhisk infrastructure enter through NGINEX.
4) Then, NGINEX forwards the request to the controller that works as the custody of the system, this one realizes the authentication and the authorization of the request before handing the control to the next component. The included credentials in the request are verified with the call to the CouchDB database.
5) As next step, the controller works as load balancer, verifying the invokers state. The load balancer, knowing which invoker are available, chooses one of them to invoke the requested action.
6) Kafka, "high-performance publishing and subscription messaging distribution system", allows communication between the controller and invokers where the HTTP request answer to the user with an ActivationId, the user will use this later to have access to the results of the resize image function.
7) As last step, OpenWhisk stores the function call results in the CouchDB database. In addition to the resized image, the system stores the metadata and the execution time, the type of the "cold or hot" system startup, start and end date, among others.

III. EXPERIMENTAL EVALUATION

Apache OpenWhisk was evaluated by building functions with the Java [22] and JavaScript [24] runtimes. These two programming languages were chosen because they are the most popular in GitHub [26], and according to the TIOBE index [27] they are in the top positions of the ranking. Experiments were done in the Amazon Elastic Compute Cloud (Amazon EC2), in an instance Debian stretch amd64 of type t2.medium. OpenWhisk runs on Kubernetes version 1.11 over Docker using the kubeadm-dind-cluster project [28]. The eastern region of the USA (Ohio) were used.
For the execution of the experiments the following questions were posed: What is the default performance of OpenWhisk for applications with dependencies section III-B? How OpenWhisk is optimized for better performance section III-C? Do the OpenWhisk configuration optimizations allow to reduce hot and cold-booting latencies for applications with dependencies section III-D?

The code generated for the experiment can be found in https://github.com/asquevedos

A. Case Study: Image Resizing

Small functions of the cloud can be started quickly, since they run on previously assigned virtual machines. However, functions that require large packages or libraries are overloaded and start slowly [12]. To evaluate a real serverless application, an application was implemented that allows the resize of an image has a request suggested in [9], this was implemented in Java runtimes and JavaScript of Apache OpenWhisk.

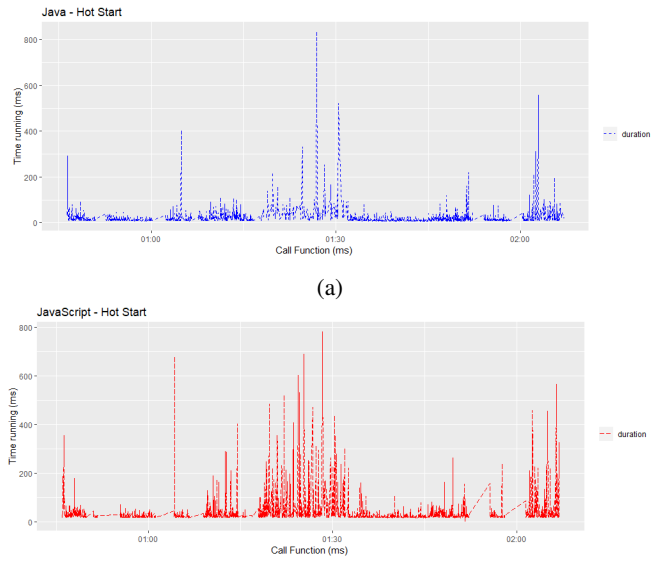

Fig. 1: Code execution duration time every 1000ms

(a) Json hot-booting. (b) Json hot-booting.

Fig. 2: Json document as a result of a function call.

B. OpenWhisk Performance

To test the performance of OpenWhisk with the default parameters, a web client is implemented that simultaneously calls the image resize functions created with Java and JavaScript every 1000ms. Figures 1a y 1b shows for ”Y” axis the code execution time; for ”X” axis shows the experiment time. In average, the execution time in Java was of 18.09ms and for JavaScript of 34.68ms. In section section III-C extends the results of the execution with the cold-booting.

C. OpenWhisk Optimization

Apache OpenWhisk allows to make changes to the configuration parameters of the system limits,including how much memory an action can use and how many action invocations are allowed per minute. In the experiment multiple changes of the default parameters are made looking for improvements in the performance of Apache OpenWisk, such as ”timeout, memory, minuteRate, concurrent” [29]. These changes according to available hardware resources can transform into improvements such as performance drops of the platform. For this experiment, with the hardware resources described in this section, the best performance is presented. The modified parameters are ”memory[1]” from 256 to 512MB and ”actionsInvokesPerminute[2]” from 60 to 200. the other parameters do not provide performance improvements in the context of the experiment.

On each call of a system function, the Apache OpenWisk architecture stores a document in Json format, which contains technical information of the execution. To measure the results of hot-booting and cold-booting this is filtered by fields of interest. The image 2a shows the Json of a hot-booting, in 2b it is observed that the executed function cold-booting thanks to the key label: ”initTime”, which shows how long the function in cold-booting is delayed. In the context of the experiment, an intentional cold start was not forced, so cold-booting is used for the measurements, which are managed and prioritized by the system.

figure 3a shows a box diagram with the obtained result of the code execution time in hot-booting with the default parameters ”Default” Vs. the best configuration parameters ”Best”. On the other hand, figure 3b, with the same approach shows the results of the cold-booting.

There is a significant change in response latencies, especially in cold-booting, the following section section III-D describes these results in more detail.

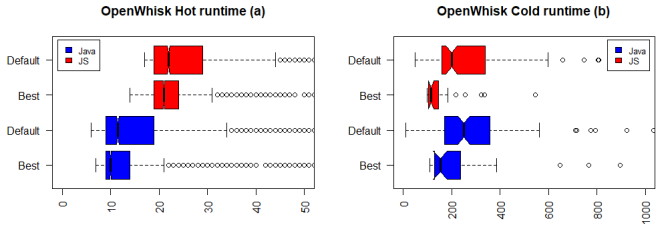

Fig. 3: Diagram box with lantencies between ”Default” vs ”Best” configuration.

D. Optimization Performance

In this section, the results obtained for the configuration of ‘’DEFAULT” parameters Vs. the ‘’BEST” configuration of parameters are formally analyzed. To compare the data, they are processed in cumulative distribution function (CDF), and

[1]A container is not allowed to allocate more than N MB of memory
[2]Limits the number of action invocations in one minute windows.

the cold-booting time of the function was 90% of the time depending on the used language.

1) Java: Figure 4 shows the obtained result of the resize image function using Java, where with the default parameters, 90% of times the cold-booting latency is less or equal to 750ms, while with the changed parameters, 90% of times the cold-booting is less or equal to 350ms, getting an improvement of 400ms.

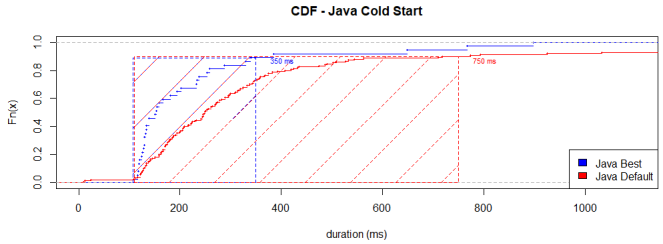

Fig. 4: Cumulative distribution function (CDF) of latencies in cold-booting between ”DEFAULT” and ”BEST” configuration using Java.

2) JavaScript: Figure 5 shows the obtained result of the resize image function using Java, where with the default parameters, 90% of times the cold-booting latency is less or equal to 900ms, while with the changed parameters, 90% of times the cold-booting is less or equal to 330ms, getting an improvement of 570ms.

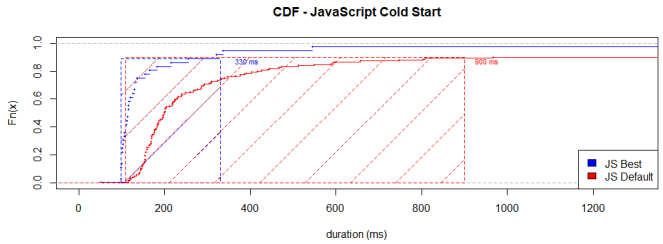

Fig. 5: Cumulative distribution function (CDF) of latencies in cold-booting between ”DEFAULT” and ”BEST” configuration using JavaScipt.

To support the results, a nonparametric Mann-Whitney U test is performed [30], since the data obtained don’t have a normal distribution. In Mann-Whitney U test the null hypothesis assumes that the two samples come from the same distribution and as an alternative hypothesis that the two samples come from different distributions. In this case for java p-value = 0.0002351 and for JavaScript p-value = 1.876e-10, so the null hypothesis is discarded and confirming that the changes made in the parameters allowed an improvement in the performance of Apache OpenWhisk.

IV. Conclusions

The changes of the default configuration parameters of Apache OpenWhisk, considering the hardware resources available, allow to improve the performance in response latencies of access to the functions that have external libraries, to execute the logic written by the programmer. In the experiment proposed in this paper, an average of 38% improvement in performance is obtained for cold-booting with functions written in the Java and JavaScript languages of Apache OpenWhisk Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Acknowledgment

This work is done thanks to the guidance and advice of Dr. Cristina L. Abad.

References

[1] *Google Cloud*. URL: https : / / cloud . google . com / functions/ (visited on 02/07/2019).

[2] *AWS — Lambda*. URL: https://aws.amazon.com/es/lambda/ (visited on 02/07/2019).

[3] *OpenWhisk – IBM Developer*. URL: https://developer.ibm . com / open / projects / openwhisk/ (visited on 02/07/2019).

[4] *Azure Functions*. URL: http : / / azure . microsoft . com (visited on 02/07/2019).

[5] David Strauss. “The Future Cloud is Container, Not Virtual Machines”. In: *Linux J.* 2013.228 (Apr. 2013). ISSN: 1075-3583.

[6] Ricardo Koller and Dan Williams. “Will Serverless End the Dominance of Linux in the Cloud?” In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. HotOS ’17. New York, NY, USA: ACM, 2017, pp. 169–173. ISBN: 978-1-4503-5068-6. DOI: 10.1145/3102980.3103008.

[7] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. “SAND: Towards High-Performance Serverless Computing”. In: *USENIX Annual Technical Conference*. 2018.

[8] “SOCK: Rapid Task Provisioning with Serverless-Optimized Containers”. In: *2018 USENIX Annual Technical Conference* (2018), pp. 57–70. ISSN: 01685597.

[9] Erwin van Eyk, Alexandru Iosup, Cristina L. Abad, Johannes Grohmann, and Simon Eismann. "A SPEC RG Cloud Group's Vision on the Performance Challenges of FaaS Cloud Architectures". In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ICPE '18. New York, NY, USA: ACM, 2018, pp. 21–24. ISBN: 978-1-4503-5629-9. DOI: 10.1145/3185768.3186308.

[10] Blesson Varghese and Rajkumar Buyya. "Next generation cloud computing: New trends and research directions". In: *Future Generation Computer Systems* 79 (2018), pp. 849 –861. ISSN: 0167-739X.

[11] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "Serverless Computation with OpenLambda". In: *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association, 2016.

[12] Cristina L. Abad, Edwin F. Boza, and Erwin van Eyk. "Package-Aware Scheduling of FaaS Functions". In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ICPE '18. New York, NY, USA: ACM, 2018, pp. 101–106. ISBN: 978-1-4503-5629-9. DOI: 10.1145/3185768.3186294.

[13] *Apache OpenWhisk*. URL: https://openwhisk.apache.org/ (visited on 02/07/2019).

[14] *NGINEX*. URL: https://www.nginx.com/ (visited on 02/07/2019).

[15] *kafka*. URL: https://kafka.apache.org/ (visited on 02/07/2019).

[16] *Docker*. URL: https://www.docker.com/ (visited on 02/07/2019).

[17] *Couchdb*. URL: http://couchdb.apache.org/ (visited on 02/07/2019).

[18] *Apache OpenWhisk Documentation*. URL: https://openwhisk.apache.org/documentation.html (visited on 02/07/2019).

[19] *Kubernetes*. URL: https://kubernetes.io/ (visited on 02/07/2019).

[20] *Apache Mesos*. URL: http://mesos.apache.org/ (visited on 02/07/2019).

[21] *OpenShift*. URL: https://www.openshift.com/ (visited on 02/07/2019).

[22] *Java*. URL: https://www.java.com/en/ (visited on 02/07/2019).

[23] *Python*. URL: https://www.python.org/ (visited on 02/07/2019).

[24] *JavaScript*. URL: https://www.javascript.com/ (visited on 02/07/2019).

[25] Ioana Baldini, Paul Castro, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, and Philippe Suter. "Cloud-native, Event-based Programming for Mobile Applications". In: *Proceedings of the International Conference on Mobile Software Engineering and Systems*. MOBILESoft '16. New York, NY, USA: ACM, 2016, pp. 287–288. ISBN: 978-1-4503-4178-3. DOI: 10.1145/2897073.2897713.

[26] *Matt Weinberger. The 15 Most Popular Programming Lan-guages, According to the 'Facebook for Programmers'*. URL: https://www.businessinsider.com/the-9-most-popular-programming-languages-according-to-the-facebook-for-programmers-2017-10 (visited on 02/07/2019).

[27] *TIOBE Index*. URL: https://www.tiobe.com/tiobe-index/ (visited on 02/07/2019).

[28] *kubeadm-dind-cluste Proyect*. URL: https://github.com/apache/incubator-openwhisk-deploy-kube/blob/master/docs/k8s-dind-cluster.md (visited on 02/07/2019).

[29] *OpenWhisk, System limits*. URL: https://github.com/apache/incubator-openwhisk/blob/master/docs/reference.md (visited on 02/07/2019).

[30] J. Russell and R. Cohn. *Mann Whitney U*. Book on Demand, 2012. ISBN: 9785511122977.