Allocation Priority Policies for Reliable Serverless Scheduling Performance

Giuseppe De Palma[1], Saverio Giallorenzo[2], Jacopo Mauro[2],

[1] Università di Bologna
[2] University of Southern Denmark

1 Introduction

Serverless computing [10] is a new development paradigm where programmers write and compose stateless functions, leaving to Serverless infrastructure providers the duty to manage their deployment and scaling. Hence, the "less" refers to the absence of some server-related concerns, namely, their maintenance, scaling, and expenses of their sub-optimal management (e.g., idle servers). Recent works investigated the Serverless paradigm under the lenses of Software Engineering, associating it to that of stateless Microservices architectures [11]

While Serverless providers have become more and more common [2, 1, 12, 5, 9, 7, 8] the technology is still in its infancy and there is much work to do to overcome the many limitations [7, 3, 6, 10] that hinder its wide adoption. One of the main challenges to address is how should Serverless providers schedule the functions on the available computation nodes, known as *function-execution scheduling optimisation* [7]. To tackle the problem, we propose a methodology that provides developers with a declarative language, called *Allocation Priority Policies* (APP) for specifying scheduling policies for their functions. Then, the serverless function-execution scheduler follows those policies to find the computation node—called *worker*—that, given the current status of the system, best fits the constraints for the given function. To substantiate our proposal, we extend the scheduler of Apache OpenWhisk [1], a well-known open-source Serverless platform, into a prototype that supports APP-defined scheduling policies. In Section 2 we present the APP language through an example and give details of our prototype.

2 The APP language

Current serverless platforms, like OpenWhisk, come equipped with hard-coded load balancing policies. With the APP language users specify customised load balancing policies and overcome the inflexibility of the hard-coded load balancing ones. The idea is that both developers and providers can write, besides the functions to be executed by the platform, a policy that instructs the platform on what computation nodes preferably run each function.

As an example, consider some functions that need to access a database. To reduce the latency of accessing the data, the best option would be to run those functions on the same pool of machines that run the database. If that option is not valid, then running those functions on workers in the proximity (e.g., in the same network domain) is preferable than using workers located further away (e.g., in other networks). We use this example to illustrate the syntax and semantics of the APP language, whose script we show in Fig. 1.

The basic entities considered in the APP language are *a*) scheduling policies, identified by a *policy tag* identifier to which users can associate their functions—the policy-function association is a one-to-many relation—and *b*) the workers identified by a *worker label*—where the label identifies a collection of workers.

An APP script is a YAML [4] file specifying a sequence of policies. Given a tag, the corresponding policy includes a list of `workers` blocks, possibly closed with a `followup` strategy. A `workers` block has three parameters: a collection of worker labels, a possible scheduling `strategy`, and an `invalidate` condition. The `followup` strategy applies when all `workers` are invalid and can either point to a special policy, called `default`, or `fail`.

```
couchdb_query:
 - workers:
   - DB_worker1
   - DB_worker2
   strategy: random
   invalidate: ↩
    capacity_used: 50%
 - workers:
   - near_DB_worker1
   - near_DB_worker2
   strategy: best_first
   invalidate: ↩
    max_concurrent_invocations: 100
 followup: fail

default:
 - workers: "*"
    strategy: platform
    invalidate: overload
```

Figure 1: APP script example.

The APP script in Fig. 1 starts with the `couchdb_query` tag, used for those functions that access the database. Then, the keyword `workers` indicates the first block of *worker labels*, which identify the workers in the proximity of the database, i.e., `DB_worker1` and `DB_worker2`, associated to three parameters: the `strategy` used by the scheduler to choose among the listed `workers`, the policy that `invalidate`s the usage of a selected worker label, and the `followup` policy in case all workers are `invalidate`d. In the example, we select one of the two worker labels `random`ly and we `invalidate` their usage if the workers corresponding to the chosen label are used at more than the `50%` of their capacity (`capacity_used`). When both worker labels are invalid, the scheduler goes to the next `workers` block, with `near_DB_worker1` and `near_DB_worker2`, chosen following a `best_first` strategy—where the scheduler considers the ordering of the list of `workers`, sending invocations to the first until it becomes invalid, to then pass to the next ones in order. The `invalidate` strategy of the block regards the maximal number of concurrent invocations for each member of a given worker label—`max_concurrent_invocations`, which is set to `100`. If all the worker tags are invalid, the scheduler applies the `followup` behaviour, which is to `fail`.

The other policy tag in Fig. 1 is `default`: a special tag used to specify the policy for non-tagged functions, or to be adopted when a tagged policy has all its members invalidated, and the `followup` option is `default`.

In Fig. 1, the `default` tag describes the default behaviour of the serverless platform running APP. The wildcard `"*"` for the `workers` represent all worker labels. The `strategy` selected is the `platform` default (e.g., in our prototype the `platform` strategy corresponds to the standard selection algorithm of OpenWhisk) and its `invalidate` strategy considers a worker label non-usable when its workers are `overload`ed, i.e., none has enough resources to run the function.

3 Our Presentation

In our presentation at Microservices 2020, we will provide some introductory notions of the Serverless paradigm and an overview of the Apache OpenWhisk [1] serverless platform. Then, we will discuss the APP syntax and semantics. We will give the main technical insights on our prototype implementation developed as an extensions for OpenWhisk. Finally, we will exhibit a serverless use case combining IoT, Edge, and Cloud Computing. We will contrast, architecture- and performance-wise, the use case implemented with APP against a naïve implementation using the vanilla OpenWhisk stack which, to achieve the same functional requirements, needs three coexisting installations of the OpenWhisk platform.

References

[1] Apache openwhisk. https://openwhisk.apache.org/, 2019. Online; acc. 04/2020.

[2] AWS. Lambda. https://aws.amazon.com/lambda/. Online; acc. 04/2020.

[3] I. Baldini et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.

[4] O. Ben-Kiki, C. Evans, and B. Ingerson. Yaml ain't markup language (yaml™) version 1.1. *Working Draft 2008-05*, 11, 2009.

[5] Google. Cloud Functions. https://cloud.google.com/functions. Online; acc. 04/2020.

[6] J. M. Hellerstein et al. Serverless computing: One step forward, two steps back. In *CIDR*. www.cidrdb.org, 2019.

[7] S. Hendrickson, S. Sturdevant, E. Oakes, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Serverless computation with openlambda. *login Usenix Mag.*, 41(4), 2016.

[8] IBM. Cloud Functions. https://www.ibm.com/cloud/functions. Online; acc. 04/2020.

[9] Iron.io. IronFunctions. https://open.iron.io. Online; acc. 04/2020.

[10] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. J. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. *CoRR*, abs/1902.03383, 2019.

[11] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara. Serverless computing: An investigation of factors influencing microservice performance. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 159–169. IEEE, 2018.

[12] Microsoft. Azure Functions. https://azure.microsoft.com/services/functions. Online; acc. 04/2020.