Presented at the 1st IEEE International Workshop on Distributed and Intelligent Systems (DistInSys)
To be published in: Proceedings of 26th IEEE Symposium on Computers and Communications (ISCC 2021)

# GOFS: Geo-distributed Scheduling in OpenFaaS

Fabiana Rossi, Simone Falvo, Valeria Cardellini
*DICII, University of Rome Tor Vergata, Italy*
{f.rossi,cardellini}@ing.uniroma2.it, falvosimone22@gmail.com

***Abstract*—OpenFaaS is a popular open-source serverless platform in the academic and industrial world. Based on Kubernetes, OpenFaaS includes a simple scheduling policy that spreads functions on cluster computing resources. As such, it is not well-suited for managing latency-sensitive applications in a geo-distributed environment, where network latencies are non-negligible and negatively affect the application response time. To overcome this issue, in this paper we present GOFS (Geo-distributed Scheduling in OpenFaaS), which extends OpenFaaS with network-aware scheduling capabilities. GOFS addresses the serverless application scheduling in a geo-distributed environment by either solving a suitable integer linear programming problem or using a greedy network-aware heuristic. However, its modular architecture facilitates the integration of other custom scheduling policies. A wide set of prototype-based results shows the advantages of the proposed network-aware solutions over other benchmark scheduling policies.**

***Index Terms*—OpenFaaS, Scheduling, Geo-distributed**

## I. Introduction

The serverless computing paradigm has recently attracted ever-growing attention; it promises to simplify the development and run-time management of applications through the Function-as-a-Service (FaaS) offering. Compared to Infrastructure-as-a-Service platforms, serverless architectures provide different trade-offs in terms of control and flexibility. The developers do not need to provision and manage servers, virtual machines (VMs), or containers as the basic computational building block for offering distributed services. Instead, they can only focus on the business logic, by defining a set of functions whose composition enables the desired application behavior. Serverless platforms can significantly simplify the operational tasks of applications, such as provisioning, scheduling, and scaling. However, they exhibit several limitations, that are further emphasized in a geo-distributed environment [1]. Today, OpenFaaS is one of the most used open-source serverless platforms, due to its lightweight and extensible architecture. Based on Kubernetes, OpenFaaS simplifies the management of complex multi-functions applications. To obtain fast start-up and shutdown times, function instances run within containers. The API gateway, one of the main components of OpenFaaS, provides an interface to create, delete, modify, monitor, and scale functions. It is also in charge of accepting both external and internal requests, and routing them to the appropriate function for processing.

Recent trends extend the traditional cloud resources with edge computing resources, placed at the network edges. The resulting environment can be beneficial for latency-sensitive applications, whose response time can be reduced by moving computation closer to data sources and consumers. This emerging environment includes fundamental aspects, which are often ignored [2]. The presence of heterogeneous resources and non-negligible network delays should be explicitly taken into account while solving the so-called *scheduling problem*, which defines the mapping between the application functions and the computing resources. A careless mapping can be detrimental for the resulting application response time. In a geo-distributed FaaS environment, the functions should be allocated as close as possible to the API gateway, so to avoid performance degradation. However, using the Kubernetes scheduler, OpenFaaS spreads functions among the computing nodes, not considering their geographic distribution. To overcome this issue, in this paper, we present GOFS (Geo-distributed Scheduling in OpenFaaS), a serverless scheduling system that extends OpenFaaS to operate in a geo-distributed environment. The main paper contributions are as follows:

- We design and implement GOFS; it introduces network-aware deployment capabilities into the scheduling logic of OpenFaaS. The modularity of GOFS allows us to easily integrate custom scheduling policies.
- We integrate network-aware scheduling policies in GOFS. To address the scheduling problem in a geo-distributed environment, we propose an optimization problem formulation and a greedy network-aware heuristic.
- To evaluate the proposed policies, we run a set of experiments using a Word-count serverless application. The experimental results show the benefits of a network-aware application deployment over other benchmark scheduling solutions (i.e., Greedy First Fit, Round Robin, and the default Kubernetes scheduler).

## II. Related Work

Among the well-known Cloud services and FaaS platforms to manage serverless applications, we can find AWS Lambda, OpenWhisk and OpenFaaS. In AWS Lambda, the scheduling is addressed using a bin-packing strategy which aims at placing a new function instance on an existing active VM to maximize memory utilization [3]. OpenWhisk is a serverless platform that executes functions using containers. To manage them, it supports multiple orchestration frameworks, such as Kubernetes and Apache Mesos. OpenFaaS uses the Kubernetes scheduler to spread functions among computing nodes. All these major serverless platforms treat functions as black-boxes and assume that the computing resources are locally distributed and interconnected by means of fast communication links (i.e., geographic distribution is neglected). Some works aim to improve the performance of existing serverless

978-1-6654-2744-9/21/$31.00 

platforms (e.g., [4]–[6]). For example, Rausch et al. [6] present Skippy, an extension of the Kubernetes scheduler that includes edge-aware constraints. Such constraints can be weighted differently to achieve different objectives (e.g., minimizing execution times). However, manually tuning the weights can be challenging and may require a detailed application profiling.

So far, only few research works have specifically targeted the serverless environment, especially with regards to the scheduling issue. To better investigate the existing scheduling approaches, we consider also those proposed for container-based applications, not limiting our analysis to those specifically tailored for serveless.

Existing scheduling policies rely on different methodologies, such as mathematical programming, machine learning, and greedy heuristics. Mathematical programming approaches exploits methods from operational research in order to solve the scheduling problem (e.g., [7]–[9]). Das et al. [9] present an optimization problem to dynamically decide where to execute tasks in a geo-distributed infrastructure, trying to optimize execution time and billing costs. The scheduling problem is known to be NP-hard, so other efficient approaches are needed. In recent years, machine learning has become a widespread approach to determine the application scheduling (e.g., [10], [11]). Pinto et al. [10] propose an approach, which relies on Multi-Armed Bandit, to decide the function-node mapping. Although conceptually easy to design, machine learning techniques may suffer from slow learning rate. To overcome this issue, different heuristics have been proposed, as surveyed in [2]. The most popular approaches resort to greedy heuristics (e.g., [12]–[14]). They can be easily implemented and often work sufficiently well, especially when the problem size grows. Faticanti et al. [12], for example, propose a throughput-aware greedy heuristic to allocate applications on the available computing resources.

All of the above approaches neglect the presence of the API gateway, which is a key component in serverless platforms (e.g., OpenFaaS and OpenWhisk) to support communication. Conceptually, we would like to allocate all functions as close as possible to it, so to quickly exchange traffic. Hence, in this paper we propose GOFS, an extension of OpenFaaS that introduces network-aware deployment capabilities.

## III. GOFS Prototype

### A. Kubernetes and OpenFaaS

A *pod* is the smallest deployment unit in Kubernetes. It consists of one or more tightly coupled containers that are co-located and scaled as an atomic entity. When a new pod is created, Kubernetes triggers the scheduler to identify a suitable hosting node. The default Kubernetes scheduler is *kube-scheduler*, which spreads pods on cluster resources. As such, it is not well-suited to place pods in an geo-distributed environment and to deal with non-negligible network delays. OpenFaaS relies on Kubernetes for orchestrating and managing serverless functions. In particular, Kubernetes is used for service discovering, auto-scaling, pod scheduling, load balancing, and network routing. Also, each function is deployed using a pod. OpenFaaS allows to develop complex serverless applications as a composition of autonomous and decoupled functions. The application workflow is defined in an orchestration function; in such a way, the involved functions remain unaware of their composition. In OpenFaaS, the API gateway manages and scales functions, exposes them through RESTful APIs, and collects function-level metrics through Prometheus. It acts as a reverse proxy; any function can be invoked by the API gateway synchronously or asynchronously e.g., using a publish-subscribe system such as NATS Streaming). Differently from others (e.g., OpenWhisk), OpenFaaS allows to scale functions down to zero.

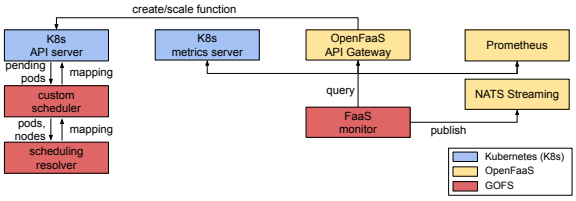


Fig. 1: High-level overview of GOFS.

### B. GOFS Architecture

Fig. 1 represents the architecture of GOFS, our extension of OpenFaaS. GOFS consists of three main components: the *custom scheduler*, the *scheduling resolver*, and the *FaaS monitor*. When new functions (pods) should be executed, OpenFaaS uses Kubernetes to allocate them on cluster nodes. To integrate novel scheduling policies within Kubernetes, we develop a custom scheduler and a scheduling resolver. The latter computes the pod allocation. Moreover, we modify the OpenFaaS source code so to use the custom scheduler, which is deployed as a pod. After retrieving the status of the cluster nodes and the list of pods to allocate, it interacts with the scheduling resolver so to identify the pod-node mapping. The scheduling resolver exposes the scheduling policy through RESTful APIs. In this paper, we rely on network-aware scheduling policies; nevertheless, other strategies can be easily integrated. Ultimately, the custom-scheduler enacts the pod to node allocation through the Kubernetes APIs. To monitor the function life cycle and the cluster health, we develop the FaaS monitor. It collects metrics from the OpenFaaS API gateway, Prometheus, and the Metrics Server. The latter is a cluster-level component that periodically collects CPU and memory utilization from all pods and nodes. Faas monitor aggregates the collected metrics and publishes them on NATS Streaming. These metrics can be then used by the scheduling resolver to analyze, at run-time, the serverless application performance.

## IV. System Model and Problem Definition

We now focus on identifying network-aware scheduling solutions for serverless applications. We consider a geo-distributed cluster shared by multiple independent applications. For each application, we assume that its functions are highly decoupled and that they communicate through the API gateway. The API gateway is deployed on a cluster node $g$ and

we do not reconfigure it at run-time. We assume that this node is correctly sized, so that the API gateway can sustain all the applications' workloads without penalizing their performance. We model the geo-distributed cluster as a graph $G = (N, E)$, where the set of nodes $N$ represents the distributed computing nodes and the set of links $E$ represents the logical connectivity between nodes. $N$ contains all the cluster nodes except $g$, the cluster node hosting the API gateway. Each node $n \in N$ is characterized by: $C_n$, the available computing resources in $n$; $M_n$, the available memory in $n$. For each link $(n, m) \in E$, we define as $d_{n,m}$ the network latency between the nodes $n$ and $m$. These attributes can be known a-priori or can be monitored and estimated at run-time.

When a client submits a serverless application to the cluster or when a new function instance is created as result of a scaling action, the cluster scheduler is triggered to accordingly identify a hosting node according to a *scheduling policy*. We denote as $\mathcal{A}$ the set of all managed serverless applications. An application $A \in \mathcal{A}$ consists of multiple functions $f$ to be allocated. Each function $f$ exposes a resource demand in terms of CPU $C_f$ and memory $M_f$. In a geo-distributed environment, we are interested in allocating functions close to the API gateway, so that they can quickly receive service requests. Therefore, each application $A$ also exposes its requirements in terms of the maximum network delay $ND^A_{\max}$ between the API gateway and each function. The scheduling policies, that we describe in Section V, explicitly take account the available computing resources, the non-negligible network delays between nodes, and the $ND^A_{\max}$ requirement while allocating the functions of each application $A$.

## V. Scheduling Policies

### A. Network-aware Scheduling Policies

In this section, we formulate the serverless application scheduling as an Integer Linear Programming (ILP) problem that explicitly models the network delays between nodes. Then, we propose a greedy network-aware scheduling heuristic to solve the problem more quickly.

*1) Optimization Problem Formulation:* We model the function scheduling with binary variables $x^A_{f,n}$, $f \in A$, $A \in \mathcal{A}$, $n \in N$, where $x^A_{f,n} = 1$ if the function $f$ of the application $A$ is placed on node $n$, and $x^A_{f,n} = 0$ otherwise. We denote the application scheduling on the computing resources with the vector $\boldsymbol{x} = \langle x^A_{f,n} \rangle$, for every $A \in \mathcal{A}$, $f \in A$, and $n \in N$.

**Node Count.** Relying on the application deployment vector $\boldsymbol{x}$, we define $Z(\boldsymbol{x})$. It counts the number of computing nodes used for running the applications. Formally, we have that:

$$Z(\boldsymbol{x}) = \sum_{n \in N} z_n \tag{1}$$

where the binary variables $z_n$ denote whether $n \in N$ hosts at least one function. We define $z_n, \forall n \in N$, as follows:

$$\frac{\sum_{A \in \mathcal{A}} \sum_{f \in A} x^A_{f,n} + \zeta_n}{\Gamma} \leq z_n \leq \sum_{A \in \mathcal{A}} \sum_{f \in A} x^A_{f,n} + \zeta_n \tag{2}$$

where $\Gamma$ is a large number and $\zeta_n$ is a constant; $\zeta_n = 1$ if $n$ already hosts at least one application function, 0 otherwise.

**Application Constraints.** Considering application-level requirements, the scheduling policy explicitly models the network delay between nodes. Note that functions communicate through the API gateway deployed on $g$. As a consequence, the scheduling policy has to allocate the functions of $A$ only on nodes $n$ whose network delay with $g$ is below $ND^A_{\max}$. Formally: $d_{g,n} \cdot x^A_{f,n} \leq ND^A_{\max}, \forall A \in \mathcal{A}, f \in A$.

**Optimal Scheduling Problem Formulation.** Our problem formulation considers an objective function that maximizes the node count. This allows us to spread serverless functions across nodes, while satisfying the $ND_{\max}$ requirements. We formulate the scheduling problem as an ILP model. Formally:

$$\max_{\boldsymbol{x}} Z(\boldsymbol{x})$$

**subject to:**

$$\sum_{n \in N} x^A_{f,n} = 1, \quad \forall A \in \mathcal{A}, \forall f \in A \tag{3}$$

$$\sum_{A \in \mathcal{A}} \sum_{f \in A} C_f \cdot x^A_{f,n} \leq C_n, \quad \forall n \in N \tag{4}$$

$$\sum_{A \in \mathcal{A}} \sum_{f \in A} M_f \cdot x^A_{f,n} \leq M_n, \quad \forall n \in N \tag{5}$$

$$d_{g,n} \cdot x^A_{f,n} \leq ND^A_{\max}, \quad \forall A \in \mathcal{A}, f \in A, n \in N \tag{6}$$

$$\frac{1}{\Gamma}\left( \sum_{A \in \mathcal{A}} \sum_{f \in A} x^A_{f,n} + \zeta_n \right) \leq z_n \quad \forall n \in N \tag{7}$$

$$\sum_{A \in \mathcal{A}} \sum_{f \in A} x^A_{f,n} + \zeta_n \geq z_n \quad \forall n \in N \tag{8}$$

$$x^A_{f,n} \in \{0, 1\} \quad \forall n \in N, A \in \mathcal{A}, f \in A \tag{9}$$

$$z_n \in \{0, 1\} \quad \forall n \in N \tag{10}$$

where (3) guarantees that each function is placed on one and only one node. Constraints (4) and (5) limit the function scheduling on a computing node $n \in N$ according to its available resources, while (6) expresses the $ND_{\max}$ constraints. Finally, (7) and (8) define the variables constraints.

*2) Greedy Network-aware Scheduling Heuristic:* Taking into account the available computing resources and the network delays between nodes, the proposed greedy network-aware scheduling heuristic (*GNet*, for short) solves a variant of the bin-packing problem (see Algorithm 1). For each $A \in \mathcal{A}$ and $f \in A$, the heuristic identifies $N^f$, the set of nodes that can host the application function $f$, also having a network delay to $g$ below $ND^A_{\max}$ (lines 9–10). If $N^f$ is empty, the application is discarded. Otherwise, the heuristic computes $\gamma_n$ for each $n \in N^f$: $\gamma_n$ approximates the number of $f$ instances that can be executed on $n$ (line 15). To balance the resource usage, the computing node with the maximum value of $\gamma_n$ is selected for allocating $f$. If multiple nodes achieve the same $\gamma_n$ value, the one closest to $g$ is selected (lines 16–17). These steps are executed repeatedly until all functions are allocated.

### B. Benchmark Scheduling Policies

In this section, we present the existing scheduling policies against which we evaluate our network-aware solutions. To-

**Algorithm 1** Greedy Network-aware Scheduling Heuristic

**Input:** $\mathcal{A}$: Serverless Applications; $N$: Set of computing nodes;
**Output:** $X$: Application scheduling;
$X = \{\}$
**for all** $A \in \mathcal{A}$ **do**
    *applicationScheduling*($A$, $N$, $X$)
**end for**

**function** APPLICATIONSCHEDULING($A$, $N$, $X$)
    **for all** function $f \in A$ **do**
        $N^f \leftarrow$ Filter $n \in N$ on $f$ resource requirements
        $N^f \leftarrow$ Filter $n \in N^f$ on $d_{g,n} \leq ND^A_{\max}$
        **if** $N^f$ is empty **then**
            discard application $A$
            **return**
        **end if**
        Compute $\gamma_n = \min(\lfloor \frac{C_n}{C_i} \rfloor, \lfloor \frac{M_n}{M_i} \rfloor)$, $\forall n \in N^f$
        $\mathcal{L} \leftarrow$ Select all nodes having maximum value of $\gamma$
        $x^A_{f,n} \leftarrow$ Allocate $f$ on the node $n \in \mathcal{L}$ closest to $g$
        $X \leftarrow X \cup x^A_{f,n}$
    **end for**
**end function**

TABLE I: Average network round-trip time (RTT) between cluster regions. Delays are expressed in milliseconds.

| | *us-central* | *eu-west* | *eu-north* | *north-america* |
|---|---|---|---|---|
| *us-central* | 1 | 100 | 135 | 32 |
| *eu-west* | 100 | 1 | 33 | 82 |
| *eu-north* | 135 | 33 | 1 | 115 |
| *north-america* | 32 | 82 | 115 | 1 |

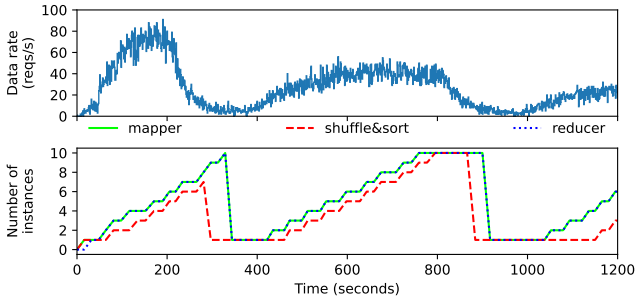


Fig. 2: Workload and number of instances for Word-count.

gether with the kube-scheduler, we include two well-known scheduling policies, namely Greedy First Fit and Round Robin.

*1) Greedy First Fit Heuristic:* It is one of the most popular solutions used to solve the bin packing problem. It considers the application functions as items to be (greedily) allocated in bins, representing the computing nodes. Specifically, the heuristic adds the available cluster nodes to a list and sorts it in ascending order of available resources.

*2) Round Robin Heuristic:* It organizes nodes in a circular list, saving the latest node used for scheduling. This heuristic allocates each function on the next node with enough resources, starting from the current position on the list.

## VI. Experimental Results

### A. Experimental Setting

We deploy GOFS on a cluster composed of 13 VMs of Google Cloud Platform; each VM has 2 vCPUs and 4 GB of RAM (type: *e2-medium*). To evaluate our solution in a geo-distributed environment, we acquire VMs in 4 different regions, where data centers are interconnected with non-negligible delays (i.e., *us-central*, *eu-west*, *eu-north*, *north-america*). The regions and the average network delays between them are reported in Table I. We acquire 2 VMs for each region, except for the central America (us-central), where we have 7 VMs. However, only 2 of them are used to schedule functions, while the other 5 VMs host OpenFaaS, the GOFS components, and the API gateway. To evaluate the proposed scheduling policies using GOFS, we implement Word-count, a surrogate serverless application. For each incoming sentence, the application extracts its words and returns the updated counter, which keeps track of the word occurrences so far received. Following the MapReduce approach, the Word-count consists of three different parallelizable functions: *mapper*, *shuffle&sort*, and *reducer*. Specifically, shuffle&sort is the orchestration function. It transfers the mappers' output to reducers, merging their outputs. As regards the shuffle&sort function, we deploy its instances using pods with 0.2 vCPU and 64 MB of RAM. For each mapper and reducer instances, instead, we use pods with 0.1 vCPU and 64 MB of RAM. The Word-count application requires $ND_{\max} = 80$ ms.

We run an extensive set of experiments aimed to evaluate the proposed scheduling policies in face of changing operating conditions. The Word-count application receives a number of requests according to the synthetic workload pattern shown in Fig. 2. To scale at run-time the Word-count application, we use the default auto-scaler of OpenFaaS. It drives elasticity by considering a statically-defined threshold on the request-per-second (RPS) metric. For the experiments, we set the static threshold to 15 RPS and a minimum and maximum number of function instances to 1 and 10, respectively. When the experiments start, all the functions have 0 instances. We observe that the scaling actions strictly depend on the workload pattern (see Fig. 2), whereas the scheduling policies do not influence scaling actions. As scheduling policies, we consider all the strategies presented in Section V and the kube-scheduler.

### B. Scheduling Policies Evaluation

To summarize the behavior of the evaluated scheduling policies, we report in Fig. 3 the overall distribution of the application response time using boxplots. Each boxplot reports the 5th, 25th, 50th, 75th, and 95th percentile of the application response time. The different policies obtain very different allocations for the application, including solutions where the functions are allocated far away from the API gateway. In such a case, the application performance is negatively affected. The Greedy First Fit, Round Robin, and kube-scheduler do not take into account network delays while computing the application scheduling; they obtain a median application response time of 529.16 ms, 567.27 ms, and 542.99 ms, respectively. Conversely, as Fig. 3 shows, the network-aware policies obtain

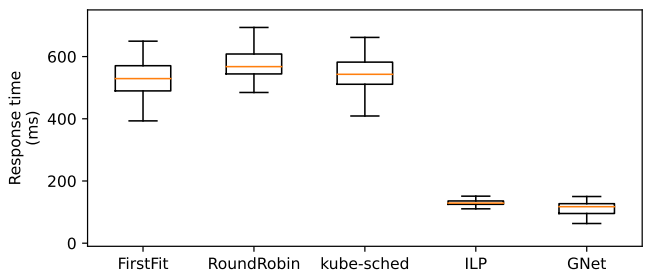


Fig. 3: Response time distribution of Word-count with the different scheduling policies. Boxplots report 5th, 25th, 50th, 75th, and 95th percentile of the response time distribution.

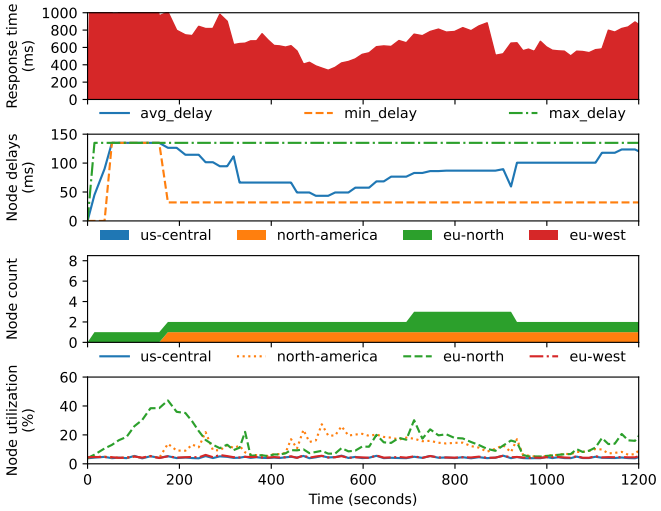


Fig. 4: Application performance with Greedy First Fit policy.

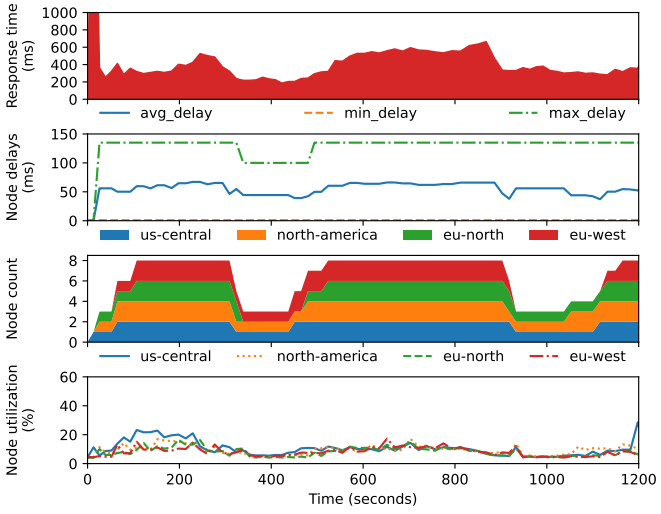


Fig. 5: Application performance with Round Robin policy.

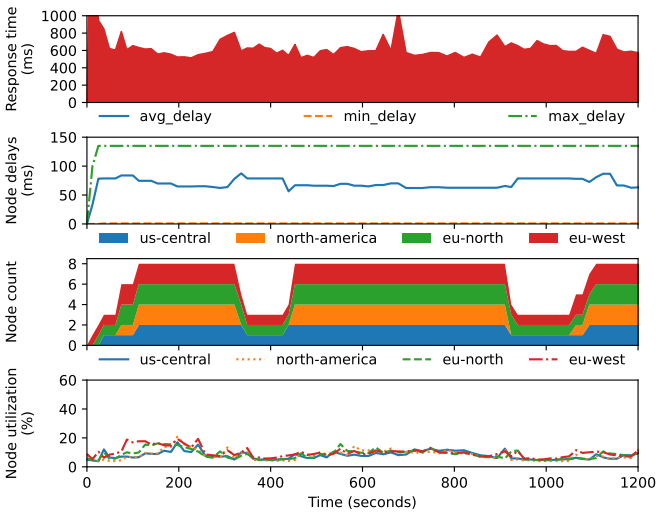


Fig. 6: Application performance with kube-scheduler.

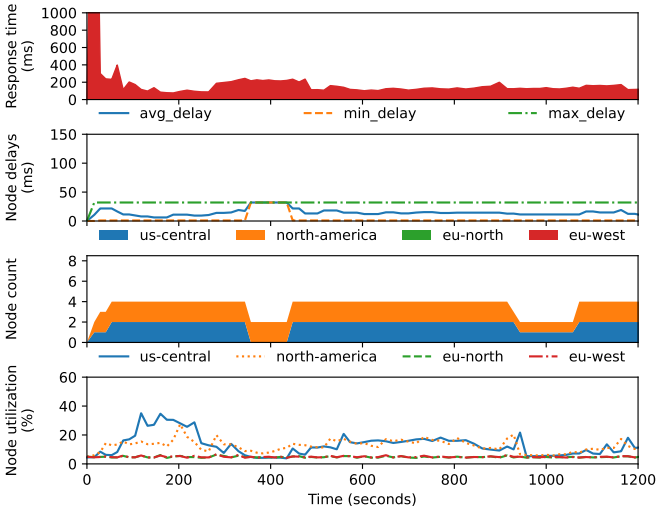


Fig. 7: Application performance with ILP policy.

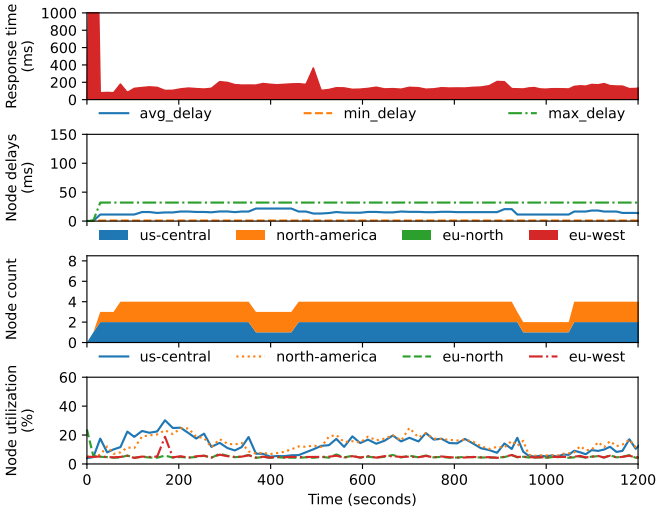


Fig. 8: Application performance with GNet policy.

a better response time distribution, with median value of $128.88$ ms for the ILP solution and $117.41$ ms for GNet. The Greedy First Fit heuristic tries to minimize at run-time the number of active nodes, placing (if possible) the newly added function pods on nodes that already host at least one pod (see Fig. 4). The application uses on average $2$ cluster nodes, spreading the functions between Europe and America. The Round Robin and kube-scheduler spread the application pods on $6$ cluster nodes. Round Robin spreads the functions across all the regions (as shown in Fig. 5), registering an average and a maximum network delay between nodes of $67.03$ ms and $135$ ms, respectively. Similarly, kube-scheduler spreads the functions between Europe and America; the maximum network delay exceeds $ND_{\max}$ most of the time during the experiment (see Fig. 6). This affects the node utilization that is, on average, equal to $8\%$. The ILP solution finds a good application scheduling, although it sometimes hosts functions in the North America region, registering a slight increase in the application response time. Anyway, it always meets the network delay requirements $ND_{\max}$, registering a significant improvement in terms of application response time compared to all previous benchmark scheduling policies. On average, the ILP scheduling policy takes $13.27$ ms to compute the allocation; this time however increases in the worst case, up to $74$ ms. Being the scheduling problem NP-hard, we expect that this time exponentially increases as the number of applications increases as well: hence, in large-scale settings, this policy may prove to be not well suited, in favor of efficient and network-aware heuristics. The GNet heuristic allows to obtain application performance close to that achieved by the ILP scheduling policy (see Fig. 8), while reducing the policy resolution time. GNet registers a maximum resolution time of $34$ ms and an average value of $10.75$ ms. This policy schedules the application pods on cluster nodes interconnected with an average network delay of $13.82$ ms and maximum delay of $33$ ms (thus satisfying the $ND_{\max}$ requirement). Also in this case, the scheduling policy deploys the application in American regions, using on average $3.46$ nodes. From Fig. 8, we observe that the node utilization by region is very close to the behaviour observed with the ILP policy. However, the GNet heuristic performs slightly better than the ILP solution in terms of application response time. When multiple nodes are suitable for hosting the newly added pods, the GNet heuristic selects always the one closest to the API gateway.

The proposed network-aware policies can be easily adjusted to minimize the number of used nodes. To this end, in the ILP formulation we need to minimize the objective function (rather than maximizing it). In GNet, we choose the first candidate node that minimizes the $\gamma$ value. In this case, the network-aware policies can allocate functions by using on average $2$ nodes and registering, on average, a reduced network delay between nodes ($14.99$ ms for ILP and $4.91$ ms for GNet).

## VII. Conclusion

The last few years have seen the increasing adoption of serverless platforms, such as OpenFaaS, to simplify the deployment of serverless applications. Based on Kubernetes, OpenFaaS has been originally designed to operate in a locally-distributed cluster, so it includes a scheduling policy that simply spreads its functions among the computing nodes. This approach is not well-suited to tackle the heterogeneity of the new emerging geo-distributed environment, such as edge computing. In this paper, we proposed GOFS, an extension of OpenFaaS that introduces network-aware scheduling capabilities. The GOFS architecture allows to employ custom scheduling policies. Therefore, we designed an optimal scheduling policy and a greedy network-aware heuristic, specifically tailored for serverless applications. The experimental evaluation showed the benefits of network-aware scheduling policies when OpenFaaS is deployed in a geo-distributed environment. As future work, we plan to extend the current heuristics modeling other performance metrics (e.g., fault tolerance, network usage, energy consumption, cold-start times), which can be of utmost importance for latency-sensitive serverless applications running on geographically distributed resources.

## References


[1] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira *et al.*, "What serverless computing is and should become: The next phase of cloud computing," *Commun. ACM*, vol. 64, no. 5, p. 76–84, 2021.

[2] V. Cardellini, F. Lo Presti, M. Nardelli, and F. Rossi, "Self-adaptive container deployment in the fog: A survey," in *Algorithmic Aspects of Cloud Computing*, ser. LNCS, vol. 12041. Springer, 2020, pp. 77–102.

[3] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proc. USENIX ATC '18*, 2018.

[4] L. Baresi and D. Filgueira Mendonça, "Towards a serverless platform for edge computing," in *Proc. IEEE ICFC '19*, 2019, pp. 1–10.

[5] Y. Xiong, Y. Sun, L. Xing, and Y. Huang, "Extend cloud to edge with KubeEdge," in *Proc. IEEE/ACM SEC '18*, 2018, pp. 373–377.

[6] T. Rausch, A. Rashed, and S. Dustdar, "Optimized container scheduling for data-intensive serverless edge computing," *Future Gener. Comput. Syst.*, vol. 114, pp. 259–271, 2021.

[7] Z. Huang, K.-J. Lin, S.-Y. Yu, and J. Y. jen Hsu, "Co-locating services in IoT systems to minimize the communication energy cost," *J. Innovation Digital Ecosyst.*, vol. 1, no. 1, pp. 47–57, 2014.

[8] D. Zhao, M. Mohamed, and H. Ludwig, "Locality-aware scheduling for containers in cloud computing," *IEEE Trans. Cloud Comput.*, 2018.

[9] A. Das, S. Imai, S. Patterson, and M. P. Wittie, "Performance optimization for edge-cloud serverless platforms via dynamic task placement," in *Proc. IEEE/ACM CCGRID '20*, 2020, pp. 41–50.

[10] D. Pinto, J. P. Dias, and H. Sereno Ferreira, "Dynamic allocation of serverless functions in IoT environments," in *Proc. IEEE EUC '18*, 2018.

[11] C. Cho, S. Shin, H. Jeon, and S. Yoon, "Qos-aware workload distribution in hierarchical edge clouds: A reinforcement learning approach," *IEEE Access*, vol. 8, pp. 193 297–193 313, 2020.

[12] F. Faticanti, F. De Pellegrini, D. Siracusa, D. Santoro, and S. Cretti, "Throughput-aware partitioning and placement of applications in fog computing," *IEEE Trans. on Netw. and Service Manag.*, vol. 17, no. 4, pp. 2436–2450, 2020.

[13] S. Pallewatta, V. Kostakos, and R. Buyya, "Microservices-based IoT application placement within heterogeneous and resource constrained fog computing environments," in *Proc. IEEE/ACM UCC '19*, 2019, p. 71–81.

[14] F. Rossi, V. Cardellini, F. Lo Presti, and M. Nardelli, "Geo-distributed efficient deployment of containers with Kubernetes," *Computer Communications*, vol. 159, pp. 161–174, 2020.