SoftwareX 21 (2023) 101290

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

P2PFaaS: A framework for FaaS peer-to-peer scheduling and load balancing in Fog and Edge computing

Gabriele Proietti Mattia *, Roberto Beraldi

Department of Computer, Control and Management Engineering "Antonio Ruberti", Sapienza University of Rome, Via Ariosto 25, 00185, Rome, Italy

ARTICLE INFO

Article history:
Received 21 September 2022
Received in revised form 21 November 2022
Accepted 5 December 2022

Keywords:
Edge Computing
Fog Computing
FaaS

ABSTRACT

In Edge and Fog Computing environments, it is usual to design and test distributed algorithms that implement scheduling and load balancing solutions. The operation paradigm that usually fits the context requires the users to make calls to the closer node for executing a task, and since the service must be distributed among a set of nodes, the serverless paradigm with the FaaS (Function-as-a-Service) is the most promising strategy to use. In light of these preconditions, we designed and implemented a framework called P2PFaaS. The framework, built upon Docker containers, allows the implementation of fully decentralised scheduling or load balancing algorithms among a set of nodes. By relying on three basic services, such as the scheduling service, the discovery service, and the learner service, the framework allows the implementation of any kind of scheduling solution, even if based on Reinforcement Learning. Finally, the framework provides a ready-to-go solution that can be installed and has been tested both on x86 servers and ARM-based edge nodes (like, for example, the Raspberry Pi).

Code metadata

Current code version	v1 (`scheduler` v1.0.0, `discovery` v1.0.0, `learner` v1.0.0)
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-22-00297
Code Ocean compute capsule	None
Legal Code License	GPLv3
Code versioning system used	git
Software code languages, tools, anoverd services used	Go, Python
Compilation requirements, operating environments & dependencies	Go 1.18, Python 3.8
If available Link to developer documentation/manual	https://p2p-faas.gitlab.io
Support email for questions	proiettimattia@diag.uniroma1.it

1. Motivation and significance

The Edge and the Fog Computing paradigms [1] arise from the need to distribute the computation among a set of nodes. In general, this necessity is a natural consequence of the application's non-functional requirements, which can regard the latency experienced by the users and service availability. The classic use case often refers to a smart city where computing nodes can be positioned in precise locations and also attached to 5G antennas [2]. In this scenario, we suppose that users are able to request services to the nearest node available. An inexorable issue that arises in this context is that we can often observe a non-negligible variation of the traffic to the nodes during the day [3]. This leads to some nodes being overwhelmed by a consistent number of requests per second (req/s that we call λ) thus the latency seen by the users for executing the service increase, and the node itself also can start to reject requests. In the meanwhile, other nodes may receive no traffic and be completely unloaded. This situation creates the necessity of designing load balancing algorithms which are able to reach a balanced load configuration by allowing the nodes to forward part of their traffic to others. In particular, we focus on cooperative strategies which allow no central entity or orchestrator, but every node, aware of its neighbours, can make decisions (that can also be based on Reinforcement Learning) independently from others by asking them for information that can regard their current load or other performance parameters. The only assumption that we make is that the scheduling decision is made per-single function execution request and therefore in an online manner. Different works

* Corresponding author.
E-mail address: proiettimattia@diag.uniroma1.it (Gabriele Proietti Mattia).

https://doi.org/10.1016/j.softx.2022.101290

in literature [4–6] are focused on the solution to this problem, but most of them only consider mathematical models and event-based simulations, and in general, real environments present many details and unexpected conditions that are very difficult to be grasped in a model of the system. For example, the operating system of the nodes may perform additional work in parallel to the execution of the service, the particular programming language used may add more execution latency due to the fact that it is compiled or interpreted and if the QoS requirement is tied to the latency this aspect can be crucial.

In this paper, we present P2PFaaS, a software framework whose objective is the practical implementation of cooperative online scheduling and load balancing algorithms generally studied only in mathematical and simulation prospectives. The key terms of the framework denomination are: peer-to-peer (P2P), which refers to the fact that each node can be considered a peer in the network who can share tasks with others without a central entity or orchestrator; and FaaS, which refers to the Function-as-a-Service paradigm that is chosen as the task model. The idea of the framework, which is built on different modules deployed as Docker containers, compensates for a lack of flexibility in modern orchestrators, like Kubernetes, which do not allow a custom definition of scheduler algorithm when multiple containers are deployed in different machines. This is essentially given by the fact that they are built for production and not for research.

The P2PFaaS framework has already been used in different works [3,7] in order to perform benchmarks of distributed algorithms in real environments. These tests required the installation of the framework both in x86 virtual machines and in ARM devices, in particular the Raspberry Pi.

The rest of the paper is organised as follows. Section 1.1 illustrates the environment that is needed for booting up the framework while Section 1.2 presents some related work. Then Section 2 describes the internal architecture of the framework, Section 3 presents some work in which the framework has been used for benchmarking scheduling and load balancing algorithms, Section 4 describes the potential impact of P2PFaaS, while the final conclusions are drawn in Section 5.

1.1. Experimental setting

The framework is written to be fully portable, indeed, it uses languages like Go and Python, which are available for all of the main architectures. For running the starting up the framework, the environment only needs to have Docker installed, then the framework will be built from the source. Regarding the hardware instead, it will suffice to have x86 machines or even ARM-based nodes, while in the former case we leave it to the user to clone the source and build the framework within every node, in the latter we instead suggest that the deployment can be efficiently done by using OpenBalena[1] framework. Once the OpenBalena is installed the P2PFaaS can be easily built for ARM and deployed to all the nodes in the set.

Once all the services of the framework are running the discovery service must be configured only at the first running of the nodes. This can be done by using the API `/configuration` at port 19000. The complete guide for installing the framework, as well as the documentation can be found on the website of the framework.[2]

[1] https://www.balena.io/open/

[2] https://p2p-faas.gitlab.io

1.2. Related work

The idea of constructing a framework for Fog or Edge computing is quite well addressed in literature. Indeed, similarly to our work, [8] proposes a platform for performing online machine learning with IoT data streams by leveraging Kubernetes for managing the containers that compose the framework. However, our operating model is different since we manage the scheduling of FaaS execution requests and it is done in an online manner. Then, OpenFaaS [9] is an open-source software framework which allows to easily implement FaaS functions but the software does not allow the customisation of the internal scheduler, which is left to the underlying Kubernetes framework. From this framework, we only used the FaaS creation process (see Section 2.1). In [10], the authors propose an extension of the OpenFaaS framework addressing the scheduling of the task in nodes that are distributed geographically, however, the work focuses on the scheduling of the services and not of the single tasks and the approach is more oriented to the Cloud computing environment than the Fog or the Edge one.

Finally, other works instead are still focused on the implementation of scheduling and load balancing algorithms but differently from our work, they only simulate the computing nodes, these simulators are iFogSim [11], FogWorkflowSim [12], YAFS [13], xFogSim [14] and FogNetSim++ [15]. Simulations can have advantages during the design of the algorithm but do not consider real environments' issues and parameters. Indeed, with our framework, researchers can assess the efficacy of the algorithms in real environments.

2. Software description

The proposed framework consists of independently developed modules. Each module has associated a code repository and it is built as a Docker container. This means that an always-alive process is associated with it. In general, it is a web server which exposes APIs routes. However, in delay-sensitive operations, web-socket pools are used. This has been shown to drastically reduce the time for creating the request since the setting up of the TCP socket and the handshaking are only done once.

The building of the framework can be done in any machine that supports Docker and even in ARM architectures for which `Dockerfile.aarch64` are given.

2.1. Software architecture

The overall architecture of the framework is shown in Fig. 1 which depicts the main modules.

- The *scheduler service* listens at port 18080 and represents the entrypoint of the framework where the clients can request the execution of a function via REST API.
- The *learner service* listens at port 19020 and implements the training and the inference of Reinforcement Learning models used by the scheduler service for making the scheduling decision.
- The *discovery service* listens at port 19000 and implements the nodes discovery.

These three services compose the core of the framework, then the user must install one or more FaaS that implement the services offered by the node. In this first version of the framework, for avoiding a significant effort on writing code which manages the service displacement (since the purpose of the framework is to focus on schedulers logic and performances), we assume, without loss of generality or validity of the results, that every

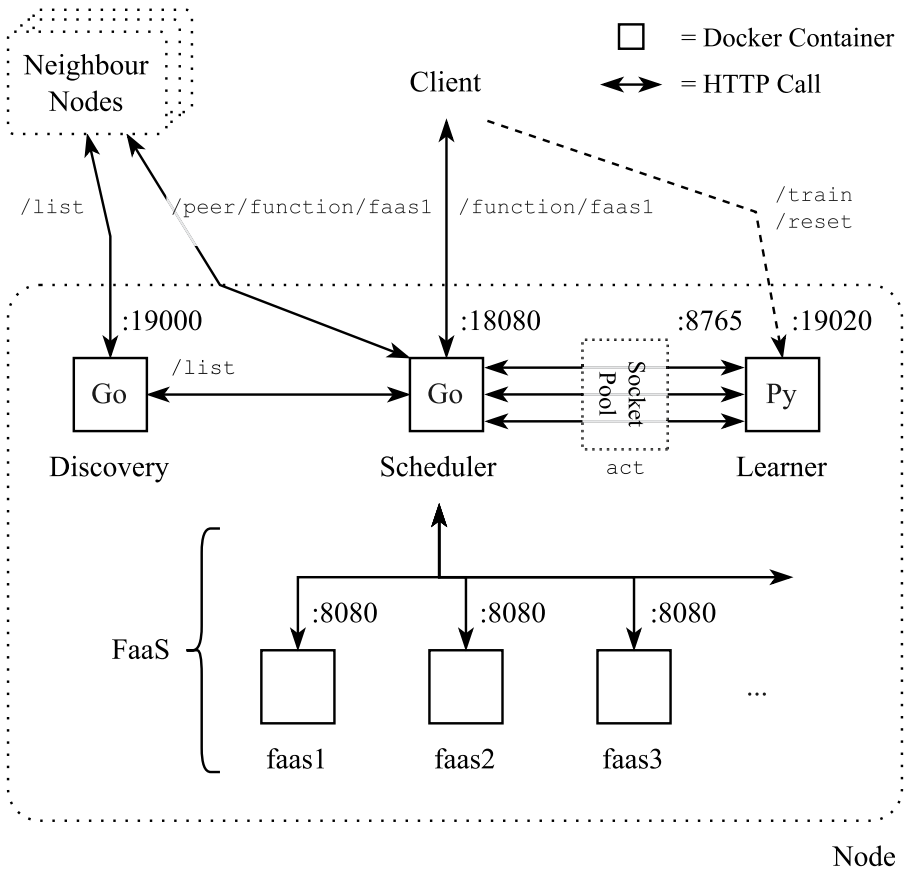

Fig. 1. The P2PFaaS high-level software architecture.

node implements the same set of functions. Therefore the framework does not provide a way for the parallel deployment of the functions; indeed, this operation must be done manually or by using OpenBalena (see Section 1.1). The functions that can be used with the framework can be borrowed from the OpenFaaS[3] open source project. It will suffice to choose a function available and packaged with the `of-watchdog`.[4] daemon and then build it with the tool `faas-cli` However, the only requirement for the FaaS is that it must be deployed as a Docker container which implements a web server that executes the function when an HTTP call is issued at port 8080 and at the root '/' route. In [3,7] the function that is used is the `pigo-openfaas`[5] function which implements a simple face recognition service.

Flow of operation. The Fig. 2 show the flow of the operations that are carried out when the client (1) requests the execution of a function (called `<fn>` in the Figure). Once the framework is set up in a set of nodes, the flow of usage starts from a client which makes a request to a node, in particular to the `scheduler` service exposes at port 18080. The URL which must be called by the client is the following:

http://ip:18080/function/<fn>

The placeholder `<fn>` must be replaced with the name of the function and it is mapped to the container name which implements the function. After making the request, the list of neighbours nodes is retrieved from the discovery service and cached. Then, a scheduling action is taken (2) and if a scheduler based on RL is configured, the current state is passed to the learner service which replies with the action to be taken. Once the action is known it is immediately executed (3) and this can require forwarding the request to another node. The request forwarding is implemented with an HTTP call to the URL:

http://remote-node-ip:18080/peer/function/<fn>

This HTTP will trigger the scheduler of the remote node and the task will be executed remotely or it can also be rejected. Otherwise, if the request has been marked as to be executed locally, the node will enqueue it and finally, it will be executed (4). The actual function execution is mapped to an HTTP call to the function's container. After the execution of the function, the output payload is finally forwarded to the client which will see its HTTP request to be concluded (5). At this point, there is an optional step that is executed only if the scheduler is RL based, that is the training of the model (6). Indeed, after the execution of the request, which is finished with the return of the output payload, we can derive the reward and forward it to the learner service which will update the weights of the model accordingly.

This concludes the operations that are needed for completing a FaaS execution request, we will now see in detail the core mechanisms of the three modules that we will call services in order to differentiate them from the sub-modules that compose them.

2.1.1. Scheduler service

The scheduler module is written in Go. The language has been chosen because it is particularly tailored for the development of web servers. Fig. 3 illustrates the architecture of the scheduler service with all the submodules that compose it.

[3] https://openfaas.com
[4] https://github.com/openfaas/of-watchdog
[5] https://github.com/esimov/pigo-openfaas

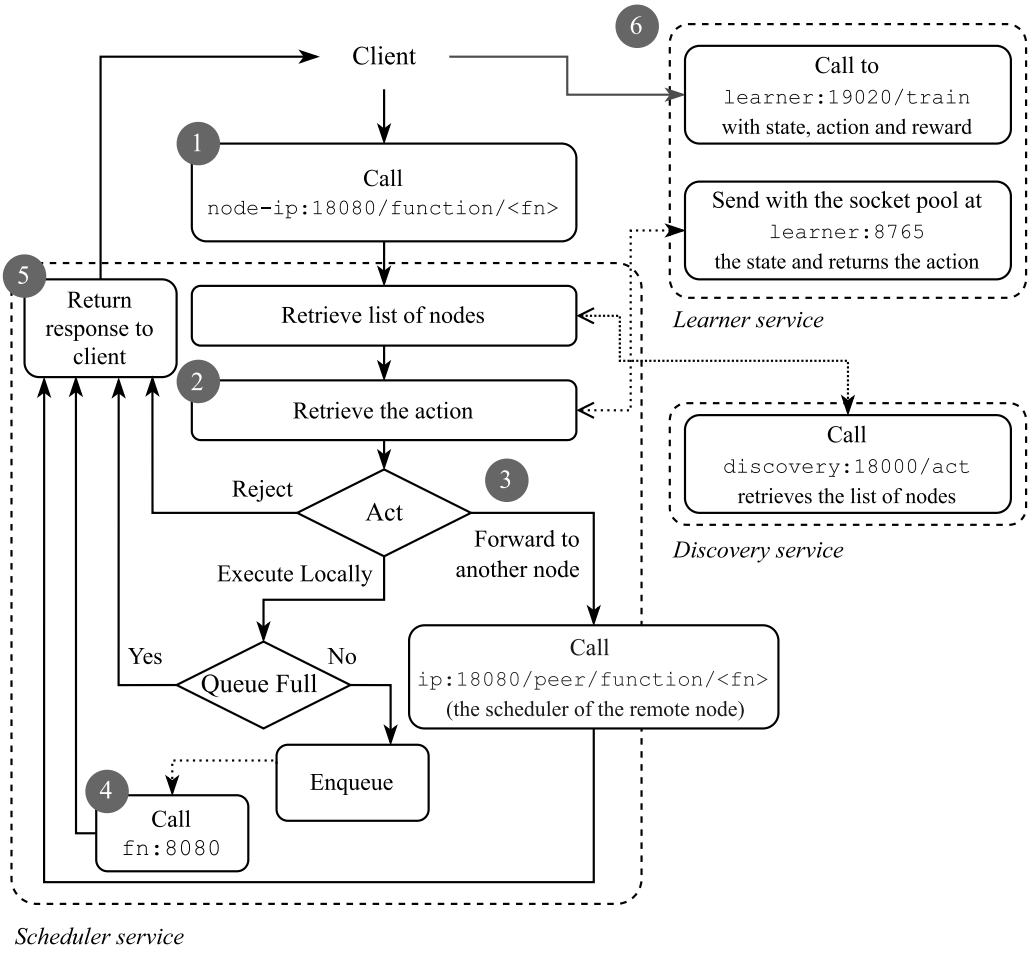

Fig. 2. The flow of the operations that takes place after a function execution request is issued from the client to the Scheduler Service.

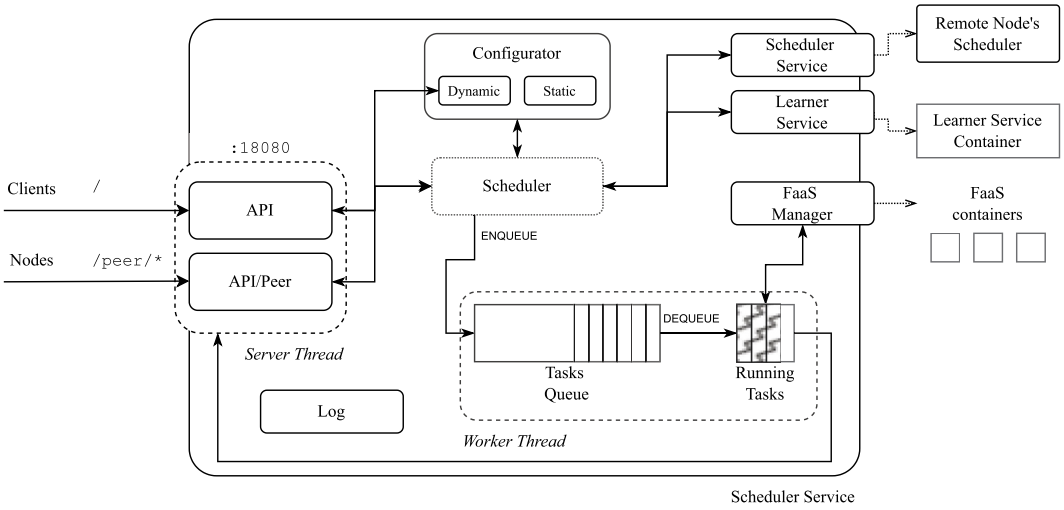

Fig. 3. The architecture of the scheduler service.

As anticipated, the scheduler service can be seen as the entrypoint of the framework. This is because the client makes the function execution request directly to it. The handling of the APIs, declared in the root Go file of the service, is done by the *API* module that handles both the preparation of the function execution request that is then forwarded to the actual scheduler and the preparation of the payload for the configuration update.

Upon the booting of the container two threads are started, one is the web server thread which manages the HTTP API calls, the other is the worker thread which manages the internal task queue.

Scheduler. The scheduler module has been designed in order to allow the interchangeability of the scheduling algorithms. For this reason, a `schedule` interface is declared (Listing). In this

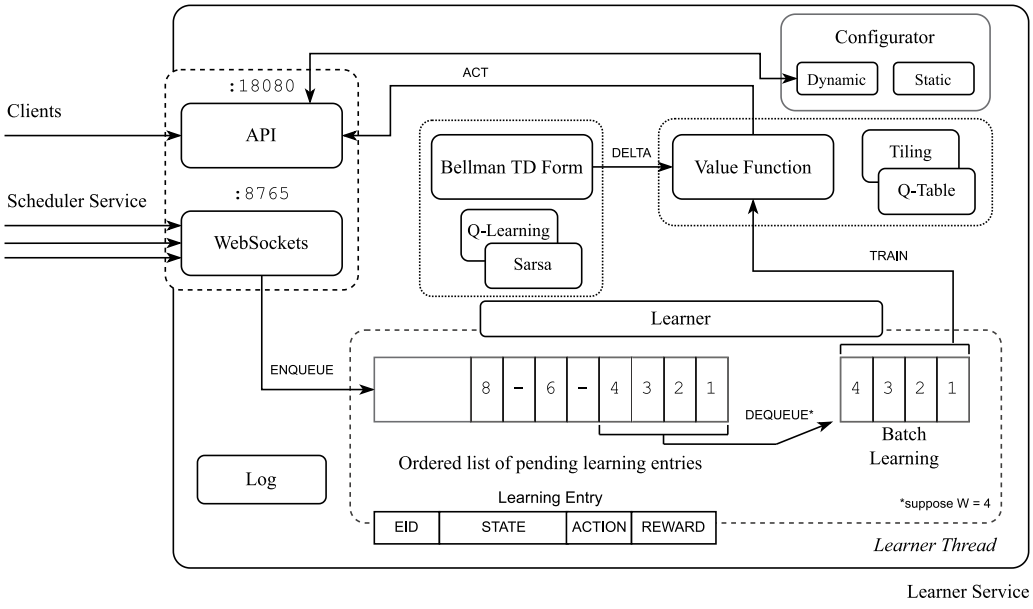

Fig. 4. The architecture of the learner service.

way, every scheduler algorithm can be easily instantiated and configured.

```
type scheduler interface {
  GetFullName() string
  GetScheduler() *types.SchedulerDescriptor
  Schedule(req *types.ServiceRequest) (*JobResult, error)
}
```

Listing 1: The scheduler interface declares the `Schedule()` function which implements the scheduling algorithm.

The `Schedule()` function implements the effective scheduling of the function execution request and three possible actions can be taken: the request is rejected, the request is executed in the current node or otherwise it can be forwarded to other nodes. When the request is rejected, the client HTTP request is immediately closed by returning the HTTP error code 500, otherwise, in other cases, the request is enqueued locally or remotely. The forwarding of the request, which is done by using the "API/Peer" (Fig. 3) module, again uses the function `Schedule()` for scheduling the request, but this time the same request will be marked as "External".

Queue. The internal queue of the scheduler has been conceived with the idea of limiting the number of parallel running functions. The parameter that is often referred to as the number of parallel tasks that can be executed in a node is called K. When, for example, $K = 4$ we are assuming that the maximum number of parallel running FaaS functions is 4. The queue is managed by a thread which implements the producer–consumer scheme, in this way a new function request is executed only when at least one running slot is available. The queue can be also limited in size, and in this case, when it is full, the requests are automatically rejected. The queue that is implemented in the described way limits the parallelism and specifically targets Edge devices which do not have a relevant computational power and at the same time it matches with models which are based M/M/1/K and M/M/K/K queues.

Faas manager. The FaaS Manager module is in charge of forwarding the function execution request to the correct FaaS function container. The module is conceived for allowing a further level of decoupling in order to allow different FaaS container technologies. The name resolution that translates the name of the container to the IP address of the FaaS container is done automatically by Docker.

Other modules. The remaining modules are the "Configurator" module which manages both a boot-time configuration (called "static") and a runtime configuration (called "dynamic"), and the "Log" module which is in charge to manage the logging, indeed extensive logging may slow down the service and increase the latency of the tasks, then we have the "Scheduler Service" and the "Learner Service" modules which both are in charge of allowing the interoperability between the Scheduler Service and the other services.

2.1.2. Learner service

The learner module is written in Python. The language has been chosen because it is widely used for machine learning. The role of the module is of implementing Reinforcement Learning models which are used for making scheduling decisions. Fig. 4 illustrates the overall structure of the service.

The learning process. Reinforcement learning models need three fundamental entities for operating: the state, the actions and the reward. The state is encoded as a string and in general, it contains the current load of the node, the action is mapped to a scheduling action and can be to execute the task locally, reject it or forward it to another node. Finally, the reward drives the learning process and it can also depend on the total task duration. For example, we may assign a positive reward if a task is completed within a certain deadline. For implementing this paradigm, we need to make the clients able to train the model, because the final delay is only known when the output payload of the function reaches the client. For this reason, the "API" module implements the `/train` and `/train_batch` routes, the first for the training of a single entity and the second for multiple entities at a time.

The learning entities. The training of the model is carried out by passing to the learner thread learning data wrapped in structures called "learning entities". A learning entity contains a progressive number (called "EID"), the state (as a string), the action (as a float) and the reward (as a float).

The learning thread. The learning process is carried out by the learner thread which is in charge to defer the training upon the fact that all the needed entities are present. Indeed, the training process must follow the specific order according to which tasks are generated by the client but it may happen that tasks did not complete in the same order as the one in which they are generated. For this reason, to each arriving task to the Scheduler Service, a progressive number is attached, and then, after the action is taken, that number (EID), the state and the action are transferred (through HTTP headers) to the client which finally triggers the training. The learning thread for deferring the learning implements a producer–consumer scheme has been implemented in such a way the training only starts when a W learning entries with consecutive EIDs are in the queue which is continuously sorted.

The learning model. The weights associated with the learning model are updated by the "Value Function" module which implements the approximation of the $Q(s, a)$ [16] function. Both the Q-Table and the Tiling methods are implemented but the framework can easily be extended even with Deep Neural Networks (DNNs). The Value Function model updates the weights according to the error that is computed by the "Bellman TD Form". This module, given the current state, the action, the next state, the next action and the reward, returns the δ. For example, the Sarsa learning strategy for the average reward is [16] shown in Eq. (1).

$$q_*(s, a) = \sum_{r,s'} p(s', r|s, a)\left[r - \max_{\pi} r(\pi) + \max_{a'} q_*(s', a')\right] \tag{1}$$

The time differential form allows knowing the error to be applied. Eq. (2) shows the time differential form of Eq. (1) when using the Q-Table for approximating the $Q(s, a)$ function.

$$\Delta_t = [R_{t+1} - \bar{R}_{t+1} + Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{2}$$

The Δ_t is returned by the "Bellman TD Forms" module and finally applied by the "Value Function" module by using the Q-Table as in Eq. (3).

$$Q(S_t, A_t) \leftarrow Q(S, a) + \alpha \Delta_t \tag{3}$$

2.1.3. Discovery service

The discovery module is written in Go, and its purpose is to allow the nodes to know which are their neighbours. The service is based on a gossip algorithm and must be configured at boot with the IP of another node (called "init server"); then, when another node, suppose B, requests to it, node A, the list of all the nodes that it knows, the IP of B will be added list nodes known by A. For now, only fully connected topologies are supported by the framework and therefore, if every node is initialised with the same init server, then every node will eventually be aware of each other.

3. Illustrative examples

Examples of the running framework have been illustrated in [7] and in [3]. In particular, [7] shows an early version of the framework running a benchmark on a power-of-n choices-based algorithm for distributed load balancing that follows a randomised approach. Instead, in [3] the framework has been used to show in practice how a Reinforcement Learning based approach for making the scheduling decision can be used on real devices. Indeed, after testing the solution in a simulated environment, the framework has been installed on 12 Raspberry Pi 4 and a Sarsa-based RL strategy has been used.

All the scripts used for running the benchmarks have been published as open source. They are available in the repository called `experiments`.[6]

4. Impact

The P2PFaaS framework presented in this work is probably the first framework available as open source which allows the implementation of distributed scheduling and load balancing algorithms between nodes by following a fully decentralised (peer-to-peer) scheme. Indeed, its flexibility is the maximum possible achievable since the development of the framework started from the constraints imposed by well-known production frameworks. P2PFaaS does not have the same level of maturity as them but for researchers in the field, it can allow testing if the designed algorithms can have a possible implementation in real devices and under which conditions they can work. Moreover, after defining the FaaS function, the scheduler can be easily written within the core of the scheduler service and changed.

Due to the portability of the code, P2PFaaS is also easy to be deployed in multiple SoC computers (like Raspberry Pis) by leveraging OpenBalena and therefore avoid using virtual machines in order to test the algorithms on real computer devices which can be bought in bulk due to their affordable cost. Testing this kind of algorithm in real devices has a clear impact on the research and, in particular, on the algorithm design. A series of conditions and peculiar characteristics of real environments cannot be easily grasped by simulations and mathematical models. For example, in the original design of the Learner Service, the Scheduler Service had to ask for the action of the Learner by using HTTP calls. However, these HTTP calls added a fixed delay of about 10 ms to each request. When testing a deadline-based scheduling algorithm this is revealed to be a critical issue, indeed, the RL-based approach was not able to outperform even a simple randomised approach. This led to the replacement of the HTTP calls with a pool of 20 web sockets which are now used in parallel only for requesting the action to the Learner Service. Therefore, mathematical models and simulations can give a direction about the performance of the algorithms in a world that is simplified, but they are fundamental to study the algorithms.

5. Conclusions

In this paper, we presented the P2PFaaS framework, a software suite which enables the testing and benchmarking of scheduling and load balancing algorithms among sets of real nodes. We showed the essential characteristics of the framework which are modularity, portability and the possibility of easily changing the core scheduler algorithm by using the designed interface. We also presented in detail the three services which compose the framework that are: the Scheduler, the Learner and the Discovery service. Further improvements of the framework are oriented to two essential directions. The first regards improving the management of service displacement by supporting dynamic functions allocations and availability. Then, the second direction regards green edge computing, indeed further modules will be added and they are currently in development to allow the benchmark of scheduling and load balancing algorithms that can make decisions considering the energy aspect of the nodes.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Gabriele Proietti Mattia reports article publishing charges and equipment, drugs, or supplies were provided by Sapienza University of Rome.

[6] https://gitlab.com/p2p-faas/experiments

Data availability

Code and data are all available as open source

Funding

This work has been supported by Sapienza University of Rome under the project "FogAware" (Grant ID: PH120172B230B4D7).

References

[1] Iorga M, Feldman L, Barton R, Martin MJ, Goren NS, Mahmoudi C. Fog computing conceptual model. Tech. rep., NIST; 2018.
[2] Pliatsios D, Sarigiannidis P, Goudos S, Karagiannidis GK. Realizing 5G vision through cloud RAN: technologies, challenges, and trends. EURASIP J Wireless Commun Networking 2018;2018(1):136. http://dx.doi.org/10.1186/s13638-018-1142-1.
[3] Proietti Mattia G, Beraldi R. On real-time scheduling in fog computing: A reinforcement learning algorithm with application to smart cities. In: 2022 IEEE International conference on pervasive computing and communications workshops and other affiliated events (PerCom Workshops). 2022, p. 187–93. http://dx.doi.org/10.1109/PerComWorkshops53856.2022.9767498.
[4] Kaur M, Aron R. A systematic study of load balancing approaches in the fog computing environment. J Supercomput 2021;77(8):9202–47. http://dx.doi.org/10.1007/s11227-020-03600-8.
[5] Alqahtani F, Amoon M, Nasr AA. Reliable scheduling and load balancing for requests in cloud-fog computing. Peer-To-Peer Netw Appl 2021;14(4):1905–16. http://dx.doi.org/10.1007/s12083-021-01125-2.
[6] Talaat FM, Saraya MS, Saleh AI, Ali HA, Ali SH. A load balancing and optimization strategy (LBOS) using reinforcement learning in fog computing environment. J Ambient Intell Humaniz Comput 2020;1–16.
[7] Beraldi R, Proietti Mattia G. Power of random choices made efficient for fog computing. IEEE Trans Cloud Comput 2020;1. http://dx.doi.org/10.1109/TCC.2020.2968443.
[8] Wan Z, Zhang Z, Yin R, Yu G. KFIML: Kubernetes-based fog computing IoT platform for online machine learning. IEEE Internet Things J 2022;1. http://dx.doi.org/10.1109/JIOT.2022.3168085.
[9] Le D-N, Pal S, Pattnaik PK. OpenFaaS. Cloud Comput Solut Archit Data Storage Implem Secur 2022;287–303.
[10] Rossi F, Falvo S, Cardellini V. GOFS: Geo-distributed scheduling in OpenFaaS. In: 2021 IEEE Symposium on computers and communications. ISCC, IEEE; 2021, p. 1–6.
[11] Gupta H, Vahid Dastjerdi A, Ghosh SK, Buyya R. IFogSim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. Softw - Pract Exp 2017;47(9):1275–96.
[12] Liu X, Fan L, Xu J, Li X, Gong L, Grundy J, Yang Y. FogWorkflowSim: An automated simulation toolkit for workflow performance evaluation in fog computing. In: 2019 34th IEEE/ACM International conference on automated software engineering. ASE, IEEE; 2019, p. 1114–7.
[13] Lera I, Guerrero C, Juiz C. YAFS: A simulator for IoT scenarios in fog computing. IEEE Access 2019;7:91745–58. http://dx.doi.org/10.1109/ACCESS.2019.2927895.
[14] Malik AW, Qayyum T, Rahman AU, Khan MA, Khalid O, Khan SU. XFogSim: A distributed fog resource management framework for sustainable IoT services. IEEE Trans Sustain Comput 2021;6(4):691–702. http://dx.doi.org/10.1109/TSUSC.2020.3025021.
[15] Qayyum T, Malik AW, Khan Khattak MA, Khalid O, Khan SU. FogNetSim++: A toolkit for modeling and simulation of distributed fog environment. IEEE Access 2018;6:63570–83. http://dx.doi.org/10.1109/ACCESS.2018.2877696.
[16] Sutton RS, Barto AG. Reinforcement learning: an introduction. MIT Press; 2018.