SAPIENZA
UNIVERSITÀ DI ROMA

# P2PFaaS: a Framework for FaaS Load Balancing


Facoltà di Ingegneria dell'informazione, informatica e statistica

Corso di Laurea Magistrale in Master Of Science in Engineering in Computer Science

Candidate

Gabriele Proietti Mattia

ID number 1645351

Thesis Advisor

Prof. Roberto Beraldi



Academic Year 2018/2019

**P2PFaaS: a Framework for FaaS Load Balancing**
Master thesis. Sapienza – University of Rome



This thesis has been typeset by LaTeX and the Sapthesis class.

Version: July 10, 2019

Author's email: pm.gabriele@gmail.com

*To my family*

*To my beloved Sara*

# Contents

# Chapter 1

# Introduction

In the 70s, the early ages of the technology revolution, all the computational power was gathered in single entities, and the ancestral concept of "terminal" referred essentially to a screen which only provided you a way to communicate with the big core in which an application was deployed. In that era, the computational power was infinitesimal if compared to modern appliances, and the energy required to power such systems was not insignificant.

The progressive change of the hardware manufactory methods, not only improved the quality and reliability parameters of the products but also caused the number of applications to exponentially grow and this also inexorably had an impact on the people needs. If before the single-tier development was the only way of building up an application, today we reached the possibility to miniaturize that development in the size of a smartphone. This continuous process of computational power improvement and miniaturizing of the hardware led to a progressive class-segmentation of the computational devices.

The traditional view of the computational power distribution is usually layered [23, 2, 9] or, in a more exotic approach, tiered [37] and even if this may seem another kind of classification, it is only another point of view of the same layer structure. This classical division, that usually comprehends four entities, was not designed in a single day by a person but it is the result of a slow and inevitable process that embodies technology improvement, research development and also society awareness. In the beginning, as already stated, all the computation was represented by single but big entities, their computational power was high as well as their energy consumption. This remembers us the modern **cloud**, the idea of having big computational entity still lives today but in a different form with respect to the past. Then the possibility to have personal computers at a reasonable cost introduced the need to connect them to the cloud, today we have smartphones, sensors and everything that can access to the internet, we call these **end devices** and as you know they can provide an interesting computational power but with low power consumption. In some critical applications the latency between end devices and the cloud cannot be tolerated, for this reason new trends brought a cloud-similar architecture nearer to the client, realizing in such a way the concept of **fog/edge** computing, thanks to the ease according to which is possible to build a cloud-like architecture without necessarily relying on the true cloud infrastructure. Then [37] also points out that there is

another layer of computing that is composed of all of those devices that are not powered by a battery but they are powered by other devices, like the RFID.

To summarize, the today recognized "places" in which computation can happen are:

1. the *Cloud*, that was the first in history to allow the development and the fruition of software applications;
2. the *Edge*, that represents the network layer but it is often called *Fog* since computation does not have to necessarily happen in network devices but also in some deployment that resembles a cloud architecture but is very near to the user;
3. the *End Devices*, and therefore every battery-powered device as well as sensors, small single-board computers;
4. *Intermittent-powered devices*, like RFID, devices that are powered by other devices when needed.

This subdivision is the one that is widely adopted in literature and it is essential for having clear in mind what is the context in which the framework that I will present in this thesis can be placed: the Fog.

## 1.1 Characterization of Fog Computing Architectures

The cloud computing model introduced different strategies of how services can be provided to the user [19]: the IaaS (Infrastructure-as-a-Service), PaaS (Platform-as-a-Service) and SaaS (Software-as-a-Service). These three paradigms, with the latest entry, the FaaS (Function-as-a-Service) [18], defined all the service capabilities that the cloud can offer to the client, by satisfying the different applications and companies needs both in terms of performance and in terms of costs.

However, the key problem of the cloud infrastructure is that the distance, and as well described in [37], the *network distance* of the cloud infrastructure from the client, is the bottleneck of using a cloud service and in some applications, especially real-time ones, the latency that characterizes this kind of connectivity is not admissible. It's from this issue that the need of making the cloud nearer to the user, and it is from this that the concept of *Fog* has been introduced. Actually, the computation can also be addressed in the network devices, and we do not essentially need other servers, that is the case in which we refer to the *Edge* computing. The real difference between Fog and Edge computing is that the Fog usually interacts with the cloud. An interesting depiction of the paradigms that come to play between the end devices and the cloud is done by [28], we have the **Fog**, the **Cloudlet** and the **MEC** (*Multi-Access Edge Computing*). The Fog refers to the paradigm of offloading end devices applications and usually interacts with the cloud; the cloudlet, that is also known as a "data center in a box", it is focused on providing services to end devices by relying on well-known virtualization technologies; the MEC focuses on offloading computation in mobile networks. These three computing paradigms have all in common the fact that they move the computation near to the end devices and the

fact that they are composed by multiple nodes that can offer computational power for providing a service.

In particular, in this Fog/Edge paradigm, three elements define its nature [23]: the **architecture**, that can be based on the flow of data, the control of the resources or the tenancy; the **infrastructure**, which regards hardware, software and middleware; the **algorithms** that can refer to the discovery, the benchmarking, the load-balancing and the placement. In general, when multiple nodes are involved in a Fog architecture a good load balancing algorithm can help the system to accommodate more requests because it can help to distribute wisely the load among all available nodes.

## 1.2 The 5G Case

In the picture of the 5G, there are some requirements that need a re-design of the current 4G/LTE architecture [21]. In particular this new design of mobile network is strictly in contact with the Fog environment, since it is essentially aimed to bring the MEC thanks to the new Cloud RAN (*Radio Access Network*) infrastructure [13].

The mobile access network, in particular, the "last mile" of the connectivity, is made possible thanks to base stations, that in the 4G terminology, are called eNodeB. Every base station is composed by two essential elements: the RRU (*Radio Remote Unit*) that provides the physical radio access to clients, and the BBU (*Base Band Unit*) that processes the signal from the RRUs and talks directly with the backbone.

In 5G we assist to the separation, in terms of location, of the two base station elements and specifically, in the C-RAN architecture, BBUs are grouped together in a pool, that can also be virtualized. This strategy has an impact on performance because the system can automatically adapt to the various load and distribute the power in order to satisfy the areas that are denser in terms of traffic. Aside to this novelty, by dividing the RRUs from the BBUs it's possible to think about putting in the BBUs some additional computational unit that is able to offer some cloud service. With these premises, the importance of MEC arise. The Multi-Access Edge Computing can be considered as a cloud computing platform that offers its services at the Radio Access Network (RAN). These services [38] can allow the offloading [41] of computation, caching and content delivery systems, enhancement in web performance, big data computing and applications for smart cities.

MEC allows the network provider to enable the RAN for low latency and therefore real-time services that are brought in proximity of users. This brings with it different challenges [21] since the network provider should allow developers to deploy services and therefore open the network to them, but it can solve the latency problem in a very easy way.

The MEC paradigm is a core concept of the 5G design and it seamlessly integrates itself with the Fog paradigm.

## 1.3 Motivation

With this introduction to Fog computing what emerges is that the field is currently flourishing of ideas and innovations and it opens to an incommensurable number of

applications and further development, but in this thesis, I will try to address one crucial problem in the Fog environment.

Both in the Fog, in MEC or in the Cloudlets multiple nodes may concur to achieve a single goal: providing the same service. In many applications, we could have the scenario in which a node may actually receive higher traffic with respect to other, and substantially in some cases is convenient to a node asking for forwarding the job request to another node. This decision, that a node needs to take, is typically a scheduling decision and the purpose of a scheduler is the one of balance the load in such a way the efficiency of the system is maximized. For this reason a scheduling algorithm needs to be conceived, in particular, in this thesis, we will focus on a subset of algorithms that allows a cooperative load balancing, as wisely classified in [23].

After having a full panorama of the cooperative load balancing problem in Chapter 2, I will provide the definition of a new kind of algorithm in Section 3. Then in Chapter 4 I will show how I implemented it as a piece of a completely new framework that I designed and developed also to be programmable with any other kind scheduling algorithm. In Chapter 5 I will report the benchmarks' result of the algorithm and I will draw conclusions in the last chapter.

# Chapter 2

# The Load Balancing Problem

There are different strategies for realizing a load balancing algorithm [23] between nodes: particle swarm optimization [22] can be used, a cooperative [7] philosophy can be addressed or we can solve the problem with the graph theory [29, 34].

The particle swarm optimization [24] is a well-known method for optimizing continuous non-linear functions and was discovered thanks to a simulation of a social model. In [22] such method is used for defining an SDN-based load balancing algorithm which aims to decrease latency and increase the QoS. Instead [29] uses the graph partition theory in order to design a Fog load balancing algorithm. Another approach, that is used by [34] is to use a breadth first search based algorithm which allows balancing the load while, at the same time, it allows also authenticate the nodes.

A great part of load balancing algorithms [7, 5, 6] in Fog environment uses the cooperation concept to operate. The idea of cooperating comes from the fact that, in order to have an optimal load balancing, we should have a master node that knows at any time the load status of every node. Since this would require a lot of effort in order to maintain such node and also a lot of probe messages that should be exchanged, is more convenient to rely on the fact that nodes are allowed to talk each other in order to both ask the load than jobs to be executed. In this perspective, instead of having a centralized scheduler we achieve a decentralization scheme in which every node acts as a single scheduler and all the nodes together realize a concept of a scheduler that is as much as possible near to the optimal one.

Before looking in deep in how the cooperative load balancing problem can be addressed we need to make a digression about the mathematical model which can represent a node that can make some work. After that we will analyze the currently best working cooperative strategies for load balancing.

## 2.1 System Models

The modeling of an entity or multiple entities that are capable of executing jobs falls in the field of the *queuing theory*, that studies how systems of these entities behave according to some defined parameters [25]. Queueing theory is actually a specialization of a wider class of dynamic systems called the “flow systems”, and if we think about it a server, or in general an entity which is able to receive jobs to be

executed, has as input a flow of jobs, that arrives at some rate, that we call $\lambda$ and as output another flow of jobs that are executed. Since executing a job takes time, the output flow will be characterized by another rate, that we usually call $\mu$. At this point, it is simple to agree with the fact that, if this input flow rate to this entity is higher than output rate, then the queue will grow infinitely since the system is not able to cope a workload that is higher than its possibilities.

When studying queueing entities, the queue models are referred by using a widely accepted notation, called the *Kendall*'s notation [20], that also allows us to summarize the queue parameters. A queue system is identified by:

$$A/B/m/K$$

Where:

1. $A$ is the arrival distribution. When studying a queue system, the input ready can also be *steady* or *unsteady*, it would be very reductive to study queues with only steady rates because we do not know what to expect when the rate fluctuates, for this reason, the input rate is usually modeled with a distribution, in particular, it is denoted as:

$$A(t) = P[time\ between\ arrivals < t] \tag{2.1}$$

   Some values of this parameter can be $M$ (Markovian process, so exponential distributed arrivals time), $E_k$ (Erlang with k as shape parameter), $D$ (deterministic) and $G$ (general);

2. $B$ is the service time distribution, that is usually denoted as:

$$B(x) = P[service\ time < x] \tag{2.2}$$

3. $m$ is the number of servers that process the queue;
4. $K$ is the capacity of the queue;

In our analysis, we will consider one machine model with both Markovian processes in input and output and infinite capacity, namely *M/M/1*, then the same with finite capacity *M/M/1/K*. As last model, we will see the one that is characterized by jobs that are executed in parallel by the same processor, namely the *M/M/1/K-PS* model.

### 2.1.1 *M/M/1*

The *M/M/1* system model is characterized by a single server that processes jobs at rate $\mu$, one job at a time. The arrival rate is $\lambda$ and both the arrivals than the output are modeled as a Markovian process, and this means that their distribution is Poissonian with rates, respectively, $\lambda$ and $\mu$. This is system is modeled with a continuous time Markov chain (CTMC), in which every state represents the current load of the machine[1].

[1]This process is often called a *birth-death process* since the transitions in the Markov chains are only of two types: we have the births when jobs arrives and deaths when job completes

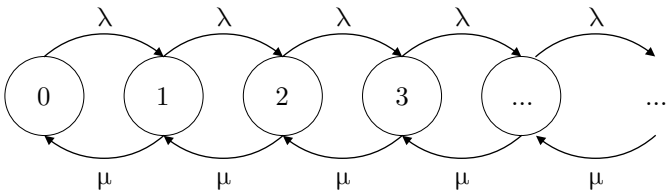


**Figure 2.1.** *M/M/1* Queue Transition States. The transition states of a *M/M/1* queue are represented by a special Continuous Markov Chains called *birth-death process*, in particular in this process birth and deaths rates are constant.

In a system like this, we are interested in two parameters: the mean number of jobs that run in the system $L$ (also called $\mathbb{E}[R]$) and mean waiting time for a job to be executed $W$ (also called $\mathbb{E}[T_R]$). Before having a look at these results we have to introduce a parameter that will be important in the next discussions, the *occupancy* $\rho$ that is defined as:

$$\rho = \frac{\lambda}{\mu} \tag{2.3}$$

This parameter gives an indication of how much the entity is loaded. In the *M/M/1* the fourth parameter in the definition is missing, this means that is $\infty$, in other words, the queue of the single entity can grow infinitely with no bounds and, from this assumption, we can easily derive that the stability condition is for $\rho < 1$, otherwise if the $\lambda$ exceeds $\mu$ than the system is not able to satisfy the demand of jobs and queue grows infinitely.

First of all, for deriving $\mathbb{E}[R]$ and $\mathbb{E}[T_R]$ we need to find the probability of the system to be in the state $i$ that can be derived by considering the flows. From the first state we have that:

$$\begin{aligned} \lambda p_0 = \mu p_1 \quad &\rightarrow \quad p_1 = \rho p_0 \\ \lambda p_1 = \mu p_2 \quad &\rightarrow \quad p_2 = \rho p_1 = \rho^2 p_0 \\ &\dots \end{aligned}$$

Therefore for a given state $i$:

$$p_i = \rho^i p_0 \tag{2.4}$$

From the total probability theorem we have that:

$$\bigcup_{i=0}^{\infty} P[\text{State} = i] = 1 = \sum_{i=0}^{\infty} p_i = \sum_{i=0}^{\infty} \rho^i p_0$$

From which we can extract $p_0$:

$$p_0 = \left( \sum_{i=0}^{\infty} \rho^i \right)^{-1}$$

Now the mean number of jobs running can be easily computed as the expected value [25]:

$$L = \mathbb{E}[R] = \sum_{i=0}^{\infty} i p_i = \frac{\rho}{1-\rho} = \frac{\lambda}{\mu - \lambda} \tag{2.5}$$

and the mean delay is [25]:

$$W = \mathbb{E}[T_R] = \sum_{i=0}^{\infty} \mathbb{E}[T_R|i] p_i = \frac{1}{\mu} \frac{1}{1-\rho} = \frac{1}{\mu - \lambda} \tag{2.6}$$

### 2.1.2 *M/M/1/K*


**Figure 2.2.** *M/M/1/K* Queue representation. This queue is characterized by a single server that processes one job at a time. The queue has a maximum length of $K$.

The *M/M/1/K* queue is a variant of the *M/M/1* but with the novelty that the queue has finite capacity $K$ and for this reason, the resulting state-transitions diagram has exactly $K$ states, as depicted in Figure 2.3.

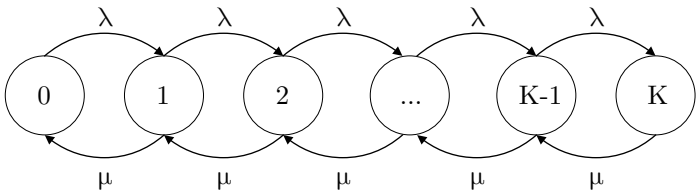


**Figure 2.3.** *M/M/1/K* Queue Transition States. If the queue has finite space, the maximum state of the machine is $K$.

Solutions to this problem are again well-known in literature, in particular here we introduce another parameter, aside $L$ and $W$, that arises due to the fact the queue is limited. Indeed, if the queue is full every new job that arrives is rejected, so we are interested in the blocking probability $P_B$, that is the probability that a job that arrives at the server finds the queue full[2], and the mean time for a job of being executed $T_R$ and this includes the execution time and the waiting time that a job encounters.

[2]That in many applications, simply means that the job is rejected

Differently from the *M/M/1* case, the solution that we will provide will be valid for any $\rho$ since the queue is limited.

We can find the blocking probability following this reasoning. The probability of the server to be in the state $i$ (in other words, the probability that the load of the machine is $i$) can be computed from the flows. From the first state we have that:

$$\begin{aligned} \lambda p_0 = \mu p_1 \quad &\rightarrow \quad p_1 = \rho p_0 \\ \lambda p_1 = \mu p_2 \quad &\rightarrow \quad p_2 = \rho p_1 = \rho^2 p_0 \\ &\dots \end{aligned}$$

Therefore for a given state $i$:

$$p_i = \rho^i p_0 \tag{2.7}$$

From the Equation 2.4 and from the total probability theorem we have that:

$$\bigcup_{i=0}^{K} P[\text{State} = i] = 1 = \sum_{i=0}^{K} p_i = \sum_{i=0}^{K} \rho^i p_0$$

From which we can extract $p_0$:

$$p_0 = \frac{1}{\sum_{i=0}^{K} \rho^i} = \frac{1-\rho}{1-\rho^{K+1}} \tag{2.8}$$

By combining Equations 2.7 and 2.8 we obtain:

$$p_i = \frac{1-\rho}{1-\rho^{K+1}} \rho^i \tag{2.9}$$

The blocking probability $P_B$ can be expressed as the probability that the system is in the state $K$, namely:

$$P_B = p_K = \frac{1-\rho}{1-\rho^{K+1}} \rho^K \tag{2.10}$$

For extracting the average waiting time for a job to be completed $W$ we can use the Little's Law which states that the average number of jobs that are executed in a server ($L$) in any given time is the product of their arrival rate ($\lambda$) and the average time that they spend in the server ($W$). Since we are only interested in jobs that are actually executed and not rejected, we need to multiply $\lambda$ for the factor $1 - P_B$ thus obtaining that:

$$L = \lambda(1 - P_B)W$$

from which we can obtain the average time:

$$W = \frac{L}{\lambda(1 - P_B)} \tag{2.11}$$

Now since the expected value of the number of jobs that are processed in a server in a given time and given the state $i$:

$$L = \sum_{i=0}^{n} i p_i \tag{2.12}$$

As a result, the formula that allows us to compute the average expected value of the waiting time is:

$$W = \frac{\sum_{i=0}^{K} ip_i}{\lambda(1-P_B)} = \frac{1-\rho}{\lambda(1-\rho^K)} \sum_{i=0}^{K} i\rho^i = \frac{1}{\mu-\lambda} - \frac{K\rho^{K+1}}{\lambda(1-\rho^K)} \tag{2.13}$$

Equations 2.10 and 2.13 are the fundamental parameters that we will use also to evaluate also a cooperative load balancing algorithm since it will be compared with the case in which the cooperation is not enabled, that is when a server represents a *M/M/1/K* queue.

### 2.1.3 *M/M/1/K-PS*: processor-sharing

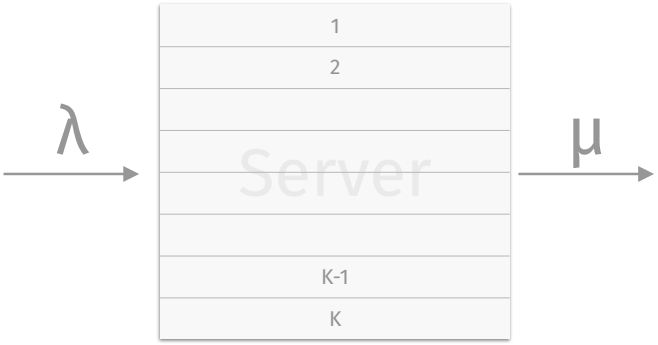


**Figure 2.4.** *M/M/1/K-PS* Queue representation. This queue is characterized by a single server that processes up to $K$ jobs in parallel and it has no queue.

On real machines, we have interleaving of processes, and this means that when a job is currently running in CPU we can accept another job but with the fact that every job will only use $\frac{1}{2}$ of the CPU time, until one of two finishes. If we suppose that a machine can have at maximum $K$ jobs that runs in parallel then in the extreme case a job can last until $K$ times its normal duration when it is the only one that runs on the CPU.

Different works found which are the distribution of sojourn times [14, 11] but theoretically and in practice (as shown in Section 5.2) if we consider $W$ and $P_B$ at the steady state, models *M/M/1/K* and *M/M/1/K-PS* are equivalent.

## 2.2 Power-of-choices algorithms

As we discussed in the introduction, cooperative load balancing implies that nodes exchange messages to each other in order to understand which is the current load that is needed in order to perform a scheduling decision. But a problem arises at this point: when a node receives a job request, does he really need to ask to every other node its load? In any case, this would be the condition that would lead us to an optimal solution but it's proven that we can reduce a lot the number of probes by choosing only a few random machines and still obtaining acceptable results.

### 2.2.1 *SQ(d)* and loss/non-loss models

The problem is commonly presented in the following way, but it can be seen also from very different points of view [27]. We have $n$ FIFO servers and customers arrive with a Poisson distribution of rate $\lambda n$ with $\lambda < 1$. Each customer chooses $d$ servers independently and uniformly at random and waits for the execution of the service at the one that has the lowest number of customers in the queue. This problem is addressed in [26] that defines this model as the **supermarket model** and the discussed algorithm as the *SQ(d)* scheduling policy. What is shown is that by choosing $d = 2$ we have an exponential improvement of waiting time of customers with respect $d = 1$, and $d = 3$ only gives a constant factor improvement with respect $d = 2$. This model is a non-loss model, this means that every job request is enqueued and never rejected. Instead [40] studies the same approach but when a server is full, meaning that a job cannot be executed in a machine that does not have enough free resource units, thus the job is rejected. This last work introduces an important performance coefficient, the **blocking probability** $P_B$, that is defined as the probability that a job request encounters a machine which cannot execute it and therefore is dropped.

In the next two sections we will see an application of these two *power-of-choices* paradigms (the [26] and the [40]) for realizing cooperative load balancing algorithms. For simplicity, we will only consider cases in which there are no reward o incentive on cooperating, but we will focus, instead, in a class of algorithms in which the cooperation is triggered only if a certain level of load is reached, this new parameter will be called threshold ($T$).

### 2.2.2 *PP(F)* and *SQ(F)*: the probing

An interesting cooperative algorithm based on the power-of-choices paradigm is presented in [5]. The work extends the idea of the *SQ(d)* (Shortest among $d$) that is the matter of [26].

In the classic *SQ(d)* algorithm, a worker entity is modeled as a *M/M/1* queue and there is a single scheduler entity that receives jobs. This approach reveals to have some down sides when deployed in a Fog environment: race conditions may arise, concurrent schedulers may assign tasks to servers that are lightly loaded. To overcome these issues, [5] proposes a protocol that is instead based on a random walk of a single probing over at most $F$ Fog nodes, the *SQ(F)* algorithm. In this way, every node is turned into a scheduler, so when a job arrives at a node it can decide whether to probe $F$ other nodes and sending the task to the less loaded or, if this does not exist, to execute the job locally.

There are two variants of this scheme, in the first *SQ(F)* the probe of $F$ node is sequentially performed, and in the second *PP(F)*, that is only an adaptation of the first, we have a parallel probing mechanism.

The *PP(F)* algorithm implements the following strategy. When a node receives a job $F$ nodes are probed. The answer of a probe message will be an ACK with the load if it is lesser than the one of the node which started the probe, otherwise NACK is returned. If a node receives all NACK from this probing phase, then it executes the job locally otherwise the task is sent to be executed to the node that reported the

| Algorithm | Workers Model |
|---|---|
| *SQ(d)* | non-Loss |
| *SQ(F)* | non-Loss |
| *PP(F)* | non-Loss |
| *LL(d)* | Loss |
| *LL(F,T)* | Loss |

**Table 2.1.** Load Balancing algorithms and their worker models

less load. This approach can be used both with Local (LR) and Remote Reservation (RR) paradigms in order to reserve servers in case of probing.

Instead in the *SQ(F)* algorithm, when a node receives a job it only sends one probe message with its load and the $f = F$ fanout parameter. Now if the probed node's load is lesser than the initiator node an ACK is returned and the job is executed in this node, otherwise the probed node increments the counter $f$ and sends a probe to another node picking it at random without the ones that have been already visited.

In both these two applications, machines are modeled with a *M/M/1* queue, this means that the queue can grow infinitely.

### 2.2.3 *LL(F,T)*: the threshold

In the previous section, we appreciated the fact that a node can act both as a scheduler than a worker. Since this double view of a node, makes sense to trigger the cooperation only when the current load of a node exceeds a certain threshold: this is the key idea of the algorithm *LL(F,T)* [8] that follows from the centralized scheduler with a lossy model of *LL(d)* [26].

First of all, when a single node receives a job it must apply a scheduling decision by probing $d$ random machines and selecting the one that is Least Loaded, we call this algorithm $LL(d)$. Whether we choose a loss or a not-loss model for the servers, this paradigm tends to be inefficient since the scheduling decision is biased towards nodes that have a low load. $LL(F, T)$ follows the same philosophy but the probing to $F$ nodes is done only if the current load of the machine is higher than a threshold $T$.

The algorithm works as follows. When a node receives a job and the current load $k < T$ then the job is executed locally, otherwise $F$ nodes are probed. If among these nodes, there is no one that has a load that is lesser than the initiator node then the job is executed locally otherwise it is forwarded to the less loaded node.

# Chapter 3

# The LL-PS(T) algorithm

In this chapter, I will present a cooperative load balancing algorithm that is based on the ones that we discussed in Section 2 but introduces a novelty regarding the system model that it uses. This novelty takes into account the fact that real systems are processor sharing, thus more than one job can be executed in parallel and every running job will spend the same proportion of CPU time (in the general case of an Egalitarian Processor Sharing system).

## 3.1 Principle of Operation

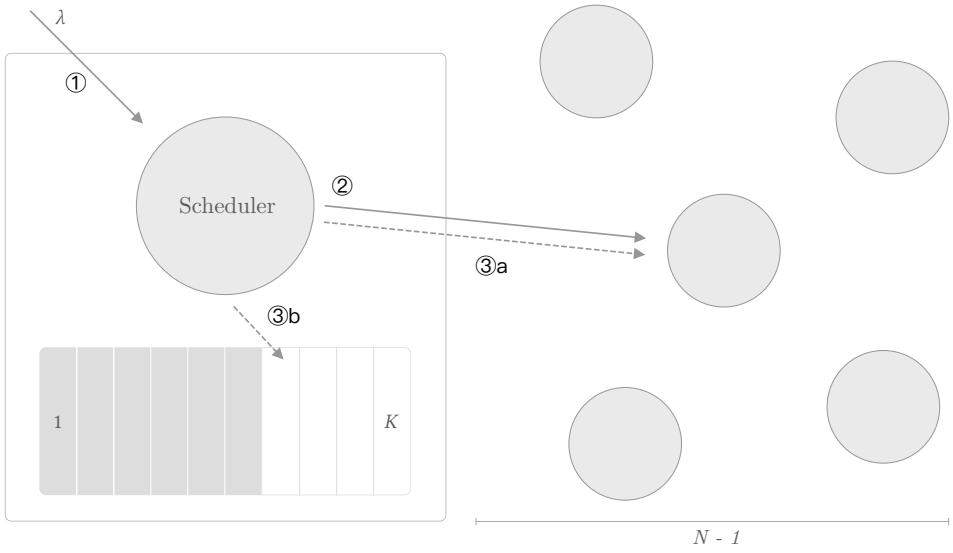


**Figure 3.1.** The *LL-PS(T)* principle of operation. Jobs arrive at a node with a Poisson distribution with mean $\lambda$ (1) and for every job the scheduler checks the current load: if it is below the threshold $T$ the job is executed locally in parallel with other running jobs with a processor sharing fashion (3b), otherwise a random node is probed (2). At this point, if there is a node that is less loaded than the current one, then the job is sent to it (3a) otherwise it is executed locally (3b).

Every node, which participates to this protocol, acts both as a worker and as job scheduler to a remote node, selected at random.

We call the node that receives a job for execution from an external mobile user, the *origin* node. Before executing a newly received job, the origin checks how many jobs it is currently running. If this value is lower than the threshold $T$ then it immediately executes the job with no other actions. Otherwise, the origin probes one node picked at random by requesting its current load (that is the number of currently running functions). At this point, if the origin manages to find a node which is less loaded than itself, then it forwards the job to such a node; otherwise, it executes the job locally. A job is dropped if both the origin and remote node are running the maximum number of allowed functions. Algorithm 3.1 shows the scheduling procedure.

**Algorithm 3.1** The *LL-PS(T)* cooperative load balancing algorithm.

**Input**: $T$ threshold, $j$ job to be executed

```
1: procedure SCHEDULE(T, j)
2:     load ← currently running functions
3:     if load < T then
4:         ExecuteLocally(j)
5:     else
6:         node ← pick one random node
7:         if Load(node) < load then ExecuteRemotely(j, candidate)
8:         else
9:             Drop(j)
```

## 3.2 System Model

First of all, the *LL-PS(T)* algorithm assumes that every node is represented by a *M/M/1/K-PS* queue, that is equivalent to the *M/M/1/K* model (this has been discussed in Section 2.1.3). The correctness of this model is assumed according to benchmarks that have been done on a single machine, see Section 5.2).

We will see an asymptotic and an approximated model that best describes the protocol and its implications. In order to perform this analysis, we will consider a large number of $N$ nodes, with $N \to \infty$. In this limit and with a random selection policy, nodes become independent from each other [12] and we then can focus on a single node. According to the experiments, the observed tagged node is modeled with a Processor Sharing queue and its internal state, that is the number of currently running jobs, is encoded with a birth-death process (Figure 3.2) where both the birth and the death rates depend on the current state $k$, as detailed next.

In the standard *M/M/1/K-PS* model, both the birth rates and the death rates are independent of the current state of the node. Here we make them strictly dependent on the current state for the following reasons.

**The birth rate** The birth rate, defined as $\lambda_k$, is actually composed of two flows of incoming jobs:

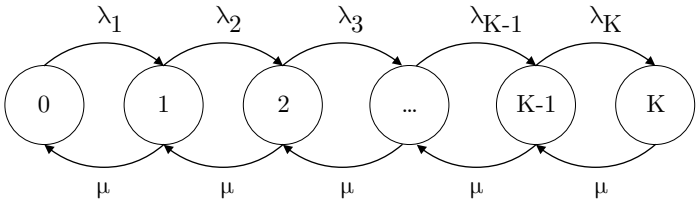


**Figure 3.2.** A node Markov chain for *LL-PS(K,T)*

$$\lambda_k = \lambda_{1k} + \lambda_{2k}$$

The first flow of jobs that determines the $\lambda_{1k}$ rate is given by the clients that make function invocation requests to the node. When such request is received by a node, the job is executed locally in two cases: if the state $k < T$ or the randomly picked node is in a state greater[1] than $k$. Therefore, the first job rate can be defined as:

$$\lambda_{1k} = \lambda \times \begin{cases} 1 & \text{if } k < T \\ \widetilde{\pi}_k & \text{otherwise} \end{cases} \tag{3.1}$$

The second flow of jobs $\lambda_{2k}$ is determined by that jobs that are forwarded by other nodes. A node can receive a job to be executed when its load is less than the origin one, but there is a condition:

- if the current state $k < T$ then the origin has to be at least in state $T$;
- if the current state $k \geq T$ then the origin has to be at least in state $k + 1$

This, translated in formulas, becomes:

$$\lambda_{2k} = \lambda\pi_k \times \begin{cases} \widetilde{\pi}_T & \text{if } k < T \\ \widetilde{\pi}_{k+1} & \text{otherwise} \end{cases} \tag{3.2}$$

**The death rate** As happens for the birth rates, even the death one depends on the state. This intuition is derived from a real implementation of the algorithm, where both probing and job forwarding are CPU time consuming.

In particular, we define as *penalty tasks*:

(A) making a probe;

(B) replying to a probe;

(C) forwarding a job;

(D) reply for a forwarded job;

[1]Given the probability $\pi_i$ to be in the state $i$, the probability of a node to be in the state $j \geq k$ is given by $\widetilde{\pi}_k = \sum_{i=k}^{K} \pi_i$

We model the execution of these tasks as they were processed by a *M/D/1-PS* queue, following the same reasoning that we made for modeling a node, but assuming that the service time distribution is deterministic, since we suppose that these penalty tasks have a fixed duration $\mu_p^{-1}$. At this point, for determining the overall service time of the fog node $\mu^{-1}$ we must take into account that after (or before, Figure 3.3) executing a job, some CPU time is spent for executing a penalty task. Therefore, this time, that derives from the queue definition ($W = \frac{1}{2}\frac{1}{\mu-\lambda}$), is added to the service time of a job $\mu_f^{-1}$ and it considers:

- as service rate $\mu_p$, the CPU time that is needed for executing a penalty task when it is the only one that runs in the processor;
- as birth rate $\lambda_p$, the expected value of the flows of activities (A), (B), (C) and (D).

Therefore:

$$\mu^{-1} = \mu_f^{-1} + \frac{1}{2}\frac{1}{\mu_p - \lambda_p} \tag{3.3}$$


**Figure 3.3.** The job execution time representation in *LL-PS(T)*

Let the probe and forward rates denoted as $\lambda_{pp}(k)$ and $\lambda_{pf}(k)$, and let:

$$\lambda_p(k) = \lambda_{pp}(k) + \lambda_{pf}(k) \tag{3.4}$$

The arrival rate of a penalty task can be defined as:

$$\lambda_p = \sum_k \lambda_p(k)\pi_k$$

As far as regards $\lambda_{pp}(k)$ we have two cases:

1. if the current state $k < T$ then the node only answer to probes of other nodes (B), so for every job it receives $N$ flows of probes at rate $\lambda$ from nodes that are at least at the state $T$, and the current node is selected with probability $\frac{1}{N}$;
2. if the current state $k \geq T$ then even the current node also needs to do a probe for every job request that it receives (A), for this reason, we add a $\lambda$ to the previous result.

This reasoning leads to define:

$$\lambda_{pp}(k) = \begin{cases} \frac{1}{N}(N\lambda)\widetilde{\pi}_T & \text{if } k < T \\ \lambda + \frac{1}{N}(N\lambda)\widetilde{\pi}_T & \text{otherwise} \end{cases}$$

which becomes:

$$\lambda_{pp}(k) = \begin{cases} \lambda\widetilde{\pi}_T & \text{if } k < T \\ \lambda + \lambda\widetilde{\pi}_T & \text{otherwise} \end{cases}$$

The forwarding penalties are applied again in two cases:

$$\lambda_{pf}(k) = \lambda_{pf_1}(k) + \lambda_{pf_2}(k)$$

The first case occurs when a job is forwarded (C), and this happens when the current state is $k$ and the state of the node to which the job is forwarded is less than $k$:

$$\lambda_{pf_1}(k) = \lambda\pi_k(1 - \widetilde{\pi}_k) \tag{3.5}$$

The second case is when a node replies with the result of a job that has been forwarded to it (D). In this case, we have that the flow of that jobs depends on the fact that the current state is $k$ and the state of the node to which the output of the forwarded job is sent is in state greater than $k$ but at least $T$, for this reason, the $\max\{k+1, T\}$:

$$\lambda_{pf_2}(k) = \lambda\pi_k\widetilde{\pi}_{\max\{k+1,T\}} \tag{3.6}$$

The steady state probabilities vector $\vec{\pi}$ of this chain is determined using a fixed point algorithm that is shown in the next section.

### 3.2.1 Convergence

The vector of the steady state probabilities $\vec{\pi}$ can be found with a fixed point algorithm. From the flows equations applied to the chain in Figure 3.2 we obtain that:

$$\pi_{i+1} = \frac{\lambda_i(\vec{\pi})}{\mu(\vec{\pi})}\pi_i \tag{3.7}$$

The algorithm that we can apply is the following:

1. start with $\vec{\pi}$ such that $\pi_0 = 1$ and the other components are 0;
2. compute the flows vectors $\lambda(\vec{\pi})$ and $\mu(\vec{\pi})$;
3. apply Equation 3.7;
4. normalize $\vec{\pi}$ so that $\sum_i \pi_i = 1$;
5. repeat from (2) until $\vec{\pi}$ converges[2].

[2]In practice we declare the convergence when $\text{dist}(\vec{\pi}', \vec{\pi}'') < 10^{-13}$

### 3.2.2 Parameters

From the steady state probability vector $\vec{\pi}$ we obtain the following parameters:

- the **blocking probability** that is the probability that the current node and the probed node are in the state $K$:

$$P_B = \pi_K^2 \tag{3.8}$$

- the **average number of jobs** executed by a node in a time unit:

$$L = \sum_i i\pi_i \tag{3.9}$$

- the **total waiting time** of a job to be completed (from the Little Law):

$$W = \frac{L}{\lambda(1 - P_B)} \tag{3.10}$$

### 3.2.3 Charts

In the following results, we will assume that $\mu_f = 0.27$ and $\mu_p = 0.05$, these values comes from a practical application of the algorithm (whose results are shown in Chapter 5).

In Figure 3.4 we can see the beneficial effect of the cooperation that results in less discarded jobs with respect to the case in which no cooperation is adopted, that is the *M/M/1/K* model. The waiting time is very similar to the non-cooperation model except when the load starts to be near the saturation level (that is when $\rho = 1$) but this is still acceptable since the algorithm manages to carry out more jobs.

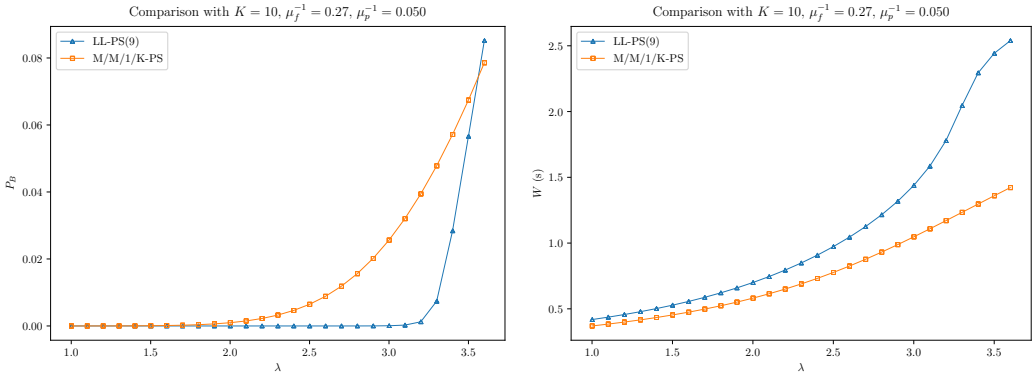


**Figure 3.4.** $P_B$ and $W$ charts for *LL-PS(9)* and $K$ set to 10.

#### Best threshold

Another result that emerges from the model is the best threshold $T$ that maximizes the efficiency of the algorithm. Due to the probing penalty that the model introduces, lowering the threshold does not correspond to increasing performances as it happens

in the model in which the penalty is not used. This because lowering the threshold implies that jobs duration is longer and therefore also the blocking probability increases since if a job lasts longer then higher is the probability that is the machine is at a higher load when a job arrives. As it will be also confirmed by practical results, the best threshold that should be used obviously depends on the ratio between the probe service time $\mu_f^{-1}$ and the job service time $\mu_f^{-1}$. We define:

$$\eta = \frac{\mu_p^{-1}}{\mu_f^{-1}} = \frac{\mu_f}{\mu_p}$$

The best threshold, that is the best tradeoff between the probe penalty and the beneficial effect of cooperation is 9 for $K = 10$, as shown in Figures 3.5.

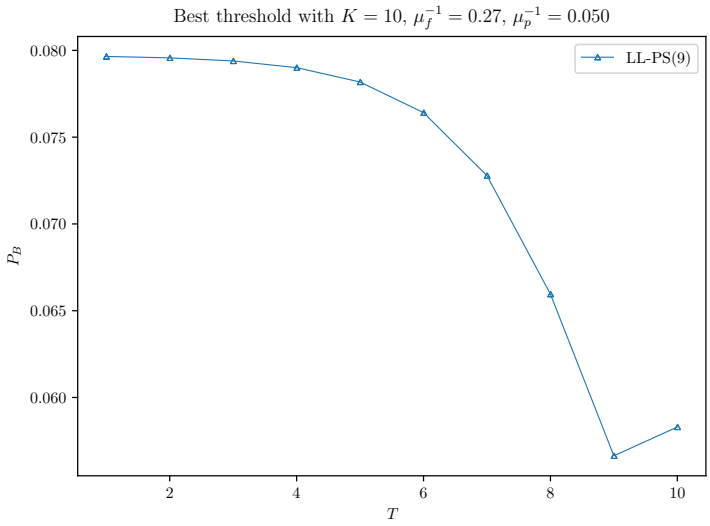


**Figure 3.5.** Best threshold chart for $P_B$ in *LL-PS(9)*

Figure 3.6 shows the delay $W$ when $\lambda = 3.50$ of *LL-PS(9)*. The decreasing behavior of the delay is justified by the fact that when the threshold is higher the cooperation is limited. By limiting the cooperation, we also limit all the penalties that are associated with it and that makes the node to persist at a higher load. As a natural consequence, if the penalties are reduced then reduced is also the average time that jobs spend in the system.

### Efficiency

By fixing $T$ and $\lambda$ we can discover the optimal $\eta$ that allows to achieve the beneficial effect of the cooperation. It is obvious if the $\mu_p^{-1}$ increases and becomes comparable with the job execution time $\mu_f^{-1}$, then make nodes to cooperate is no more convenient.

Figure 3.7 shows the *LL-PS(9)* algorithm for a fixed $\lambda = 3.50$ varying the $\eta$ ratio. What emerges is that when $\eta = 0.21$ the cooperation model becomes equal to the *M/M/1/K* as far as regards the $P_B$, therefore any higher value of $\eta$ makes the cooperation not convenient. The same does not hold for the delay $W$ which appears

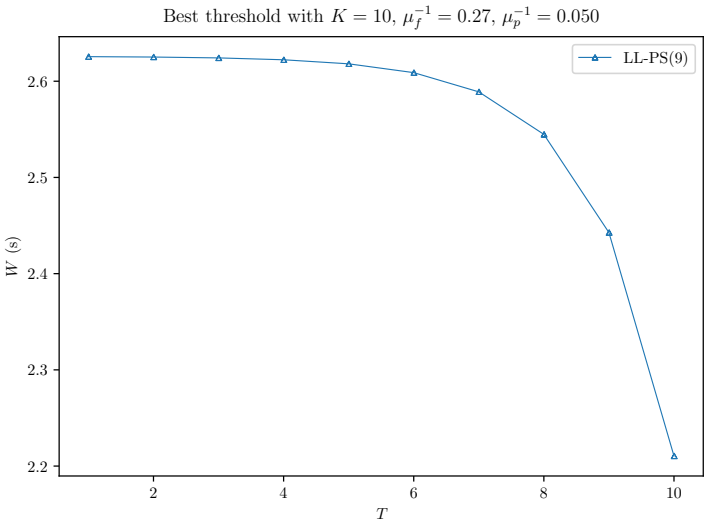


**Figure 3.6.** Best threshold chart for $W$ in *LL-PS(9)*

to be in any case higher than the non-cooperation model, but again if it is true that the delay is higher then it is also true that we are accepting more jobs.

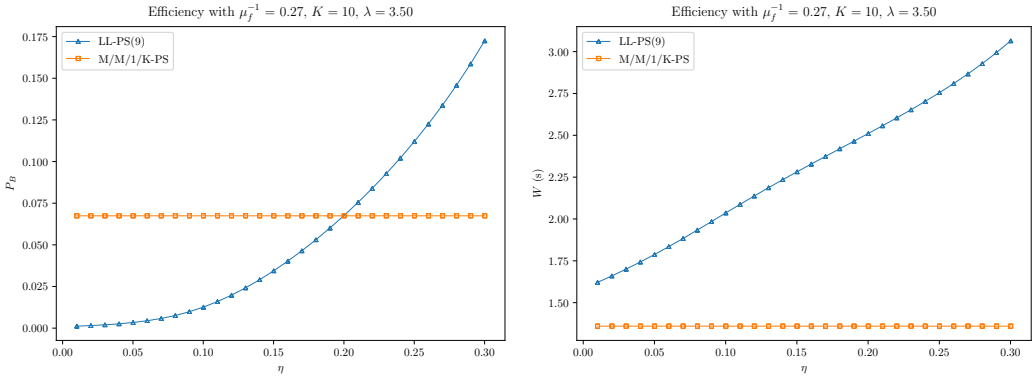


**Figure 3.7.** Efficiency chart for $P_B$ and $W$, when $K = 10$ and $\lambda = 3.50$, in *LL-PS(9)* compared with *M/M/1/10* model

In Figure 3.8 a different point of view is given for the efficiency. In this case, we have a long run with $\lambda = [2.0, 3.6]$ of *LL-PS(9)* with six different values of $\eta$. This chart again confirms what we have already discussed, but here is possible to appreciate that the higher the $\eta$ the sooner and the steeper is the "jump" that $P_B$ and $W$ have when $\lambda$ increases.

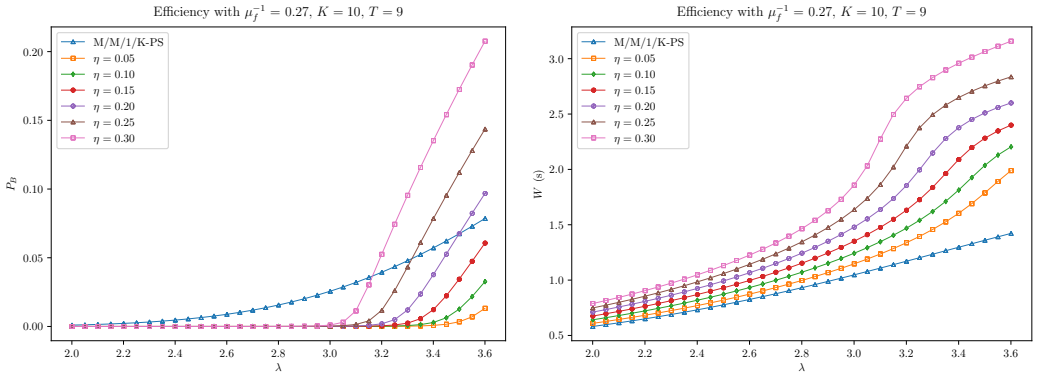


**Figure 3.8.** Efficiency chart in a long run for $P_B$ and $W$, when $K = 10$, in *LL-PS(9)* compared with *M/M/1/10* model

# Chapter 4

# Implementation

In this chapter, I will show how the framework is internally designed and realized. I will present all the internal components and the logic behind them, but only after motivating the raw materials that I make use as building blocks of the framework.

## 4.1 Materials

Before starting even to think about implementing a load balancing algorithm, it's necessary to conceive a possible job unit that clients can request to be executed and that machines can easily exchange. This will be the core data structure of the entire framework. All that follows derives from this decision.

### 4.1.1 The Function-as-a-Service (FaaS) paradigm

Function-as-a-Service is a really modern paradigm that probably in next years will replace many of those interactions that are now realized with IaaS or PaaS. In the FaaS model, the service provider offers to the user not an infrastructure or a platform (e.g. a web server, or any other kind of endless run application) but a function, for this reason, this model is also known as "Serverless computing". In general, a function can be easily moved between machines, it can be easily defined and so it requires a very low overhead of configuration and deployment.

It's clear that this model brings a great number of advantages either from the provider and the user points of view [3]. The former can easily manage the scheduling of a function, it does not need to set up an entire machine for a function but a single machine can run a high number of functions. The latter has a convenience when setting up the function since the user is not required to set up a brand new environment for only running a function, he only has to write its definition and, in many cases, declare its dependencies.

It may seem that this approach it's the solution to any kind of computational problem, but that's not true. Not all the applications fit a function-as-a-service paradigm, or only some part of it can be implemented with FaaS. Just think about a database service or a service in which we need all the core of execution is centralized. FaaS are perfect for easy applications and fast deployment, like machine learning functions, websites. Further development of the concept also provides ways of

composing functions, since there may be the case in which a single function is not enough for handling a complex problem [4].

The Function-as-a-service paradigm is the perfect candidate for implementing a cooperative load balancing scheduler, because a function is like a monad since it is independent on the specific machine that runs it, it can be easily defined, it can be easily moved between machines and so it's easy to conceive a system in which machines exchange each other functions to be executed.

### 4.1.2 OpenFaaS

There are many frameworks that allow building a FaaS infrastructure. The ones that I considered are Apache OpenWhisk and OpenFaaS but for its spirit of innovation and for the documentation available I chose OpenFaaS.

OpenFaaS [30] is an open source framework launched in 2017, whose purpose is to make serverless function simple. It can run both with Kubernetes and Docker Swarm, for its composable nature and allows the easy definition, creation, and deployment of functions in many well-known languages like Python, Javascript, C#, Go and many others.

The structure of OpenFaaS is stack-based, it exploits an underlying orchestrator for creating services, represented by functions. Despite the fact that OpenFaaS can be installed both on Docker Swarm and Kubernetes, in this framework that I am implementing, I chose to use Docker Swarm (see Section 4.1.3 for the core features) as backend, especially for its immediacy and simplicity. In particular, in P2PFaaS framework, thanks to its API endpoints, OpenFaaS is used as part of its stack (see Section 4.3).

#### Principle of function execution

Of particular interest, mainly for the implications on the model analysis (Section 3), is the principle according to which a function is executed in a Docker Swarm node (the subject matter of Section 4.1.3).

In the OpenFaaS framework, the code of a function always comes with a `Dockerfile`. A `Dockerfile` (Listing 4.1) is a declarative-type file in which Docker images are defined: the filesystem structure, the ports that it exposes and also commands that have to be executed when the image is built. OpenFaaS has a template of a `Dockerfile` for any programming language. In these files, after building and setting up the environment for the particular language, a webserver is run, called **function watchdog** [32]. The watchdog essentially provides an API gateway that forks a function *handler* at every function invocation. The function handler is a wrapper which executes the function.

```
FROM golang:1.10.4-alpine3.8 as builder

RUN apk --no-cache add curl \
  && echo "Pulling watchdog binary from Github." \
```

```
    && curl -sSL
    ↪  https://github.com/openfaas/faas/releases/download/0.9.14/fwatchdog >
    ↪  /usr/bin/fwatchdog \
    && chmod +x /usr/bin/fwatchdog \
    && apk del curl --no-cache

# language-dependent body

ENV fprocess="./handler"
EXPOSE 8080

HEALTHCHECK --interval=2s CMD [ -e /tmp/.lock ] || exit 1

CMD ["./fwatchdog"]
```

**Listing 4.1.** The `Dockerfile` of a function generated by the `faas-cli` tool. At the top of the file, is declared the command to download the OpenFaaS function watchdog and at the end the command that container will run, that is the watchdog executable.

Why we need a watchdog? Essentially for performance reasons. When using a containerization system, a container is essentially a process, so saying that the process terminated or the container terminated is the same thing, since the container is like a sandboxed running process. Thus having a new container (or a new process) that starts up when requesting the execution of a function may include some unwanted delay. For solving this problem, OpenFaaS uses its own watchdog that is bundled in the Docker image of any function that is deployed in the system. Since the watchdog is a web server, is always running and always ready to serve any execution request.

### Anatomy of a function

The deployment of a new function in OpenFaaS can be done with the command line tool called `faas-cli`. After creating a new sample function, we will have the following items:

- `Dockerfile`, the file that defines the Docker image that will contain the function and the watchdog (as mentioned in Section 4.1.2);
- `function`, a folder that contains the file `handler.go` in which we can write the body of the function;
- `main.go`, a launcher of function declared in `handler.go`;
- `template.yml` contains a set of metadata that are associated to the function, like the environment variables and the programming language;

This file structure is generated by `faas-cli` tool after building the function for the first time, indeed the action of building a function consist of calling the command `docker build` that reads the `Dockerfile` and builds the Docker image. Once the image is ready the function can be deployed with the command `faas-cli deploy` that creates a Docker Swarm service (see Section 4.1.3).

### 4.1.3 Docker Swarm

P2PFaaS makes intensive use of OpenFaaS, since it allows to deploy and run functions. For this reason, it follows the same philosophy regarding the implementation, in particular, it relies on Docker Swarm.

Docker Swarm is a Docker's *run mode* [16]. Therefore, if Docker, when running in the standard mode, allows the containerization of applications in the machine where it is installed, when instead it runs in swarm mode it becomes an orchestrator since it interacts with other machines creating a cluster.

In the swarm mode, we always have to define at least a *master node* (that is suggested to be replicated in an uneven number) which will know the state of every other node and perform orchestration operation, and *worker nodes* that will act as simply work executors.

Once we have set up the infrastructure we can deploy **services**. A service defines the task that has to be executed on nodes and is the core unit that is scheduled by the swarm orchestrator. In a simplistic view, a service is defined by a Docker image[1], and other parameters like how many replica of the service we need or if the service can restart automatically when it crashes. When a service is deployed, Docker Swarm will start it in the less loaded node, and if a replica is needed of the same service, again it will be started in the less loaded machine, but when a request to that service is received from the outside an **internal load balancer** will send that request to less loaded node[2] also by using an internal DNS (since a service must be always available at the same port in every node). In the Docker Swarm lexicon, a service is *the definition of a task* and a task is the *actual running instance of the service*.

What is important is that from the external, the entire cluster will be seen as it was a single node. This also means that if a service is deployed in a Docker Swarm, no matter where the request for that service is done, it will be always available at the same address and port, transparently to the user.

As it will be shown in Section 4.2, P2PFaaS will be deployed as two Docker Swarm services.

### 4.1.4 Go and Rest APIs

One of the most used programming languages for building distributed applications is Go. Go is an open source programming language developed by Google that is focused on simplicity, reliability, and efficiency. With Go, it's easy to write efficient software since it does not provide complex constructs and it is a compiled language.

Another key point of Go is its natural predisposition to build REST-ful services. In general, the most used approach for allowing containerized services to communicate is through REST API endpoints, therefore they expose a port in an *overlay network*[3]

[1] A Docker image is an image of a filesystem where the application that has to be executed has been deploy and it ready to run

[2] This is also known as **ingress load balancing**

[3] Also used in Docker, an overlay network is a network in which are attached containers. Docker assigns a local IP to every container, but containers are also reachable through their name like `http://container-name:port`

for example, and then all the inter-container communication happens to HTTP methods like GET and POST. This is the philosophy that is also used in P2PFaaS.

## 4.2 Design

Having gathered all the materials it's now possible to depict a possible design of how the system could be realized.

OpenFaaS offers all of its features, that includes the deploy, removal, and execution of a function by mean of its REST APIs that are exposed at the specific `8080` port of the `faas-swarm` container. The scheduler that we want to implement needs to execute functions in the machine where it is installed, therefore it will need to communicate directly to OpenFaaS that will handle this kind of task. The logical consequence of this is to deploy a **scheduler** like a Docker swarm service that will be attached to the OpenFaaS overlay network, in such a way, from the scheduler itself, OpenFaaS will be reachable at the address `http://faas-swarm:8080`.

One of the most important peculiarities of P2PFaaS is that the scheduler is distributed, every machine decides where to execute jobs but this assumes that machines know each other existence. For this reason, another service is needed, a **discovery** service that is accessed by the scheduler when it wants to know where to send jobs to be executed. A discovery service of this kind will be again deployed as a Docker Swarm service and attached to the network where the scheduler is.

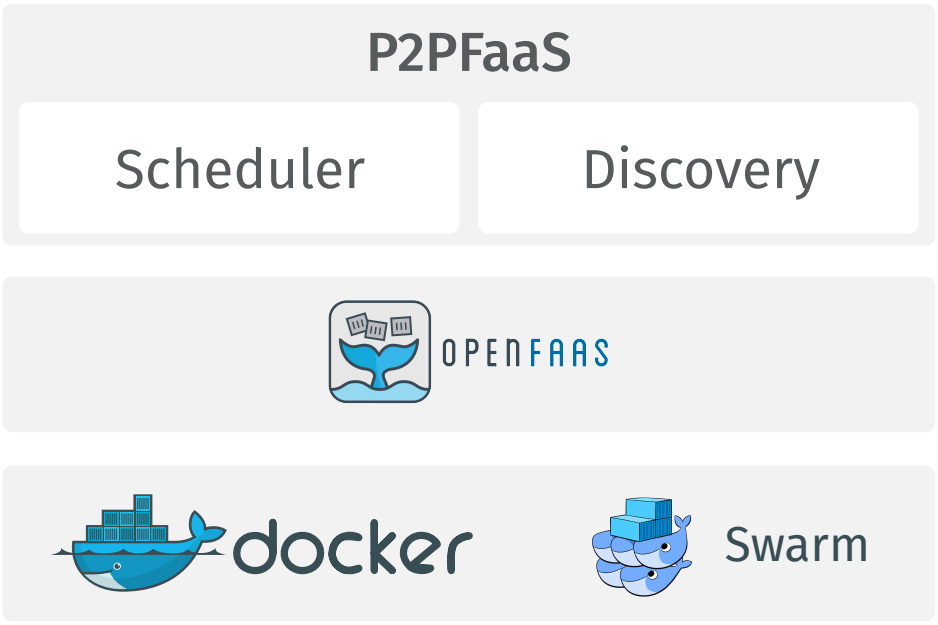


**Figure 4.1.** P2PFaaS infrastructure. P2PFaaS is lying above OpenFaaS since it uses all of its API Endpoints but at the same time, it is deployed as Docker Swarm service.

Figure 4.1 summarize the entire design of the P2PFaaS framework.

## 4.3 Scheduler

The role of the scheduler service is that of received requests to execute functions, make a decision about where to execute the function, finally executing it and return the result to the client. This logic of operation expects that:

- the entire process must be transparent to the client, which does not need to do anything more than an HTTP call as it was only OpenFaaS deployed;
- all the HTTP calls of the client must be done towards the P2PFaaS scheduler;
- the operations must be synchronous, in other words, the client is blocked until the job is completely executed[4];
- the scheduler decision must be as fast as possible in order to avoid unwanted delays that could ruin the advantage of cooperation;
- the scheduler must be able to put jobs in the queue in case the machine is full and it cannot execute other jobs in parallel. This particular feature is not implemented in OpenFaaS, in other words, when executing a function with the synchronous mode there is no limit about how many forks of the same function we can have since OpenFaaS provides an auto-scaling mechanism that replicates the function service if the number of requests per second is exceeding a certain threshold.

By considering these prerequisites it's now possible to draw a list of the main components that the scheduler needs to have. First of all, we need to interface with other services, therefore we have:

- **OpenFaaS interfacer** for using the OpenFaaS services like executing functions;
- **discovery interfacer** for using the discovery services like knowing which are the other fog nodes;
- **scheduler interfacer** for using the scheduler services of other nodes, for example for executing a function remotely.

Then we have the inner components:

1. the core **scheduler** that decides where running the job according to the installed algorithm. Moreover, the scheduler may decide to use or not the queue (point 2) because only a certain number of parallel jobs is allowed to be executed (this will be subject matter of Section 4.3.5). As it will be described in Section 4.3.5, the core scheduler is defined by an interface, in such a way it can be easily changed, even at runtime;
2. the **queue** is a producer/consumer based, and its role is the one of putting jobs that cannot be executed immediately (see Section 4.3.4);

[4]OpenFaaS also offers an asynchronous mode of operation but in the P2PFaaS case is not taken into consideration

| Endpoint | Methods |
|---|---|
| `/system/functions` | `GET`, `POST`, `PUT`, `DELETE` |
| `/system/function/<name-of-function>` | `GET` |
| `/function/<name-of-function>` | `GET`, `POST` |
| `/monitoring/load` | `GET` |
| `/peer/function/function` | `POST` |
| `/configuration` | `GET`, `POST` |
| `/configuration/scheduler` | `GET`, `POST` |

**Table 4.1.** The list of endpoints exposed by the P2PFaaS's scheduler on port `18080`

3. the **configurator** manages all the configuration parameters of the system 4.3.2;
4. the **in-memory db** it's a simple value storage for fast accessing 4.3.3.

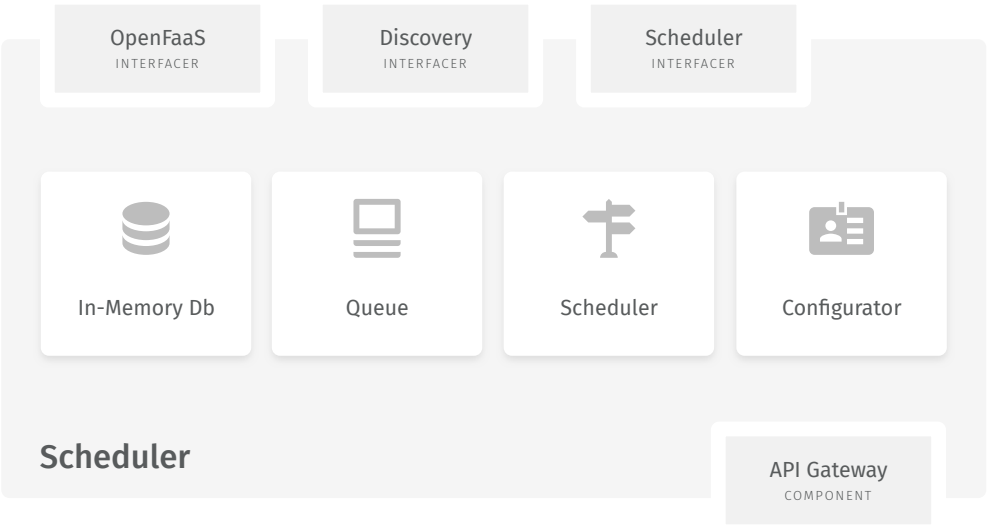


**Figure 4.2.** The P2PFaaS's scheduler architecture. Scheduler is composed of different units that are represented by Go packages.

The scheduler design is depicted in Figure 4.2 and as a final component, that allows the client to access its services, we have the **API gateway**.

### 4.3.1 API Gateway

The *API Gateway* component exposes the services of the scheduler to clients via REST APIs. The main endpoints are:

1. `/system/functions` is directly mapped to the OpenFaaS' facilities. It can be used with methods `GET`, `POST`, `PUT` and `DELETE` for manipulating functions:

| Header | Description |
|---|---|
| X-PFog-Descriptor | The name of the scheduler used |
| X-PFog-Timing-Probe-Messages | The number of probe message used |
| X-PFog-Timing-Queue-Seconds | "Queuing Time" in seconds |
| X-PFog-Timing-Faas-Execution-Seconds | "Faas Execution Time" in seconds |
| X-PFog-Timing-Execution-Seconds | "Execution Time" in seconds |
| X-PFog-Externally-Executed | If the function has been externally executed |
| X-PFog-Hops | The number of hops that the job did |
| X-PFog-Peers-List-Id | List of node IDs that handled the job |
| X-PFog-Peers-List-Ip | List of node IPs that handled the job |
| X-PFog-Timing-Probing-Seconds-List | List of "Probing Time" in second in every node |
| X-PFog-Timing-Forwarding-Seconds-List | List of "Forwarding Time" in second in every node |

**Table 4.2.** HTTP Headers used by the API Gateway component. Some of the headers are not present when the job is executed locally. Refer to Appendix A.1 for a detailed description of the timings.

with this endpoint, it's possible to deploy new functions, delete the existing ones or updating them (See [31] for further references);

2. `/function/<name-of-function>` same of the previous one, but with this endpoint is possible to retrieve information about a single function;
3. `/monitoring/load` allows getting the current load of the machine. The current load refers to the number of parallel jobs that machine is executing, and this information comes also with the number of maximum parallel jobs and the maximum length of the queue.
4. `/peer/function/function`, this endpoint must be only used by other nodes since it allows machines to send jobs to be executed and receive its output once this has been executed.
5. `/configuration` allows to retrieve the current configuration settings of the node and also to modify it (see Section 4.3.2).
6. `/configuration/scheduler` allows to retrieve the configuration of the scheduler since a scheduler can also have configurable parameters. This set configuration can be also modified at runtime (see Section 4.3.2).

### Behavior

When job a is completed, the HTTP request returns in the client and, aside of the job output that is put in the HTTP response body, the API Gateway component also injects in the HTTP headers other metadata that can be useful for analyzing the system performance. The HTTP headers that are used depends on the scheduler that is used and on the fact the job is executed in other nodes or not, however, we can draw a list of all the headers that are used in the system, see Table 4.2.

### 4.3.2 Configurator

Configuration plays an important role, especially when we are dealing with tunable algorithms and conditions. In P2PFaaS framework there is component that only deals with the configuration management. Its main features are:

1. loading of configuration from a **JSON file** in the file system at the system boot;
2. make the configuration **read-only** by the other parts of the system;
3. make the configuration **editable at runtime** only by mean of REST API and if there is no job running.

The feature (1) expects an obvious implementation, while the other two reserve more attention. In particular, the feature (2) requires a double definition of the configuration memory structure because, if it has to be read-only then the fields has not to be exported but if the fields are not exported, then we cannot read configuration from a JSON file since the un-marshaling operation requires that structure's fields are exported. For this reason, there is a double definition of the configuration structure, one that has un-exported fields, that is the one that actually saves the configuration in memory, and another one that has only exported fields, that is used for decoding JSON.

Feature (3) simply sets a new configuration by reading a JSON from the HTTP `POST` payload, then it checks if jobs are running (by mean of the in-memory DB, Section 4.3.3) and if not the configuration is saved both in memory than in the filesystem, in such a way when the system is reboot it is re-loaded.

#### Parameters

Configuration parameters that are mostly used by the framework are:

- OpenFaaS parameters, like host and port;
- discovery service parameters, like host and port;
- file path of JSON file for the configuration;
- listening port, that is `19000` by default;
- **maximum number of running functions**, that is also called $K$ and it is equivalent to the maximum number of parallel jobs that the system is able to execute;
- **maximum length of the queue**;

#### Scheduler configuration

The parameters of the scheduler, that make it tunable, follow the same philosophy of the general configuration but they are loaded in a different file and editable with a different API endpoint.

Scheduler parameters are described in the scheduler implementation sections.

### 4.3.3 In-memory db

The in-memory database is a very simple package that stores two kinds of information:

- the list of currently running functions
- the total number of functions running

The write access to this information is protected by a mutex since they are accessed in a concurrent fashion.

### 4.3.4 Queue

Another component that is included in the P2PFaaS framework is the Queue and it has been introduced for compensating the lack of this feature in the OpenFaaS synchronous mode of operation. As seen in Section 4.1.2 when a execution request is sent to OpenFaaS, the framework forks the process that is able to execute that function from the Docker Swarm service that has been deployed. This process appears to have no limits since a feature that is offered by OpenFaaS is the auto-scaling: if the number of requests is exceeding a threshold then the function service is auto-scaled.

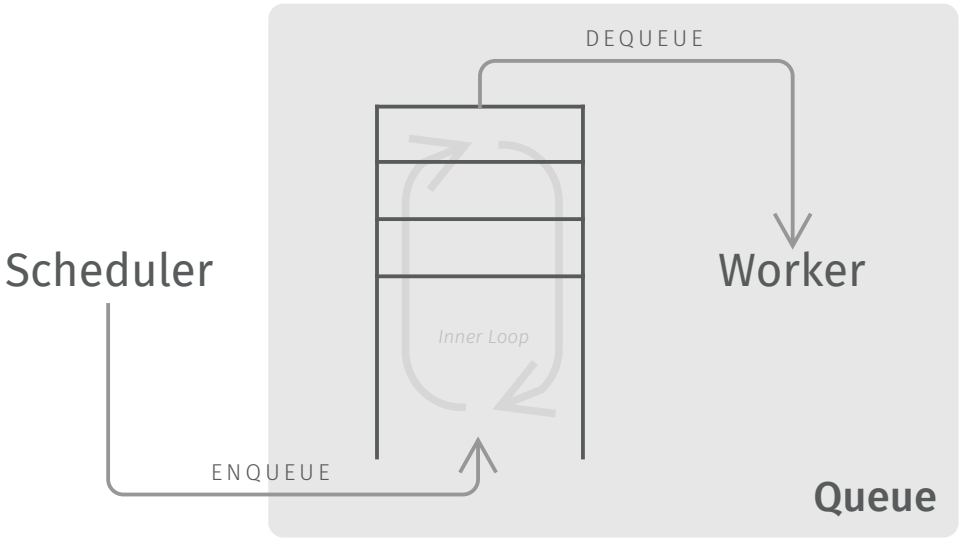


**Figure 4.3.** P2PFaaS queue. The queue is implemented with a producer/consumer fashion.

In the P2PFaaS it's possible to control how many of parallel executing functions you can have and this is achieved by using a producer/consumer scheme. When the framework is booted, a `for ; ;`[5] loop thread is started. This loop uses two semaphores, one that blocks it if there are no jobs to be executed and one that blocks it if there are no consumers to execute the jobs. As depicted in Figure 4.3:

[5]In go language this is equivalent to create a `while True` loop

- the **producer** is the scheduler component that is able to enqueue jobs;
- the **consumers** are exactly $K$, that is the number of maximum parallel jobs that can be executed.

Going in deep on the internals of this mechanism we can have a look at the main `Looper` function that is started in a thread when the framework boots up (Listing 4.2).

```
func Looper() {
  for ; ; {
    consumersSem.Wait(1)
    job := dequeueJob()
    go executeNow(job)
  }
}
```

**Listing 4.2.** `Looper` function of the scheduler's queue component

Another queue's component peculiarity is that when a job is enqueued it is wrapped into another memory structure, the one that is shown in Listing 4.3. This structure, aside from the metadata parameters, contains a semaphore that is used for telling the main loop that a consumer ended to execute the function.

```
type QueuedJob struct {
  Request   *types.ServiceRequest
  Semaphore *utils.Semaphore
  Response  *faas.APIResponse
  Timings   *Timings
}
```

**Listing 4.3.** `QueuedJob` structure in the scheduler's queue component

### 4.3.5 Scheduler

The scheduler component is the core of the service. For having a general definition of the component we can have a look at the very simple interface that describes it in Listing 4.4.

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

type Descriptor struct {
  Name       string   `json:"name"`
  Parameters []string `json:"parameters"`
}
```

**Listing 4.4.** The interface that allows to implement a scheduler is very basic since it requires only one core function.

The core function of a scheduler is the `Schedule` function. It is a blocking function that will return only when the job has been completely executed, this because it is called by the API handler for the function execution.

Let's now conceptually see which is the complete process that brings a function execution request to the effective execution. The general idea is the following (Figure 4.4) and assumes that a function $f$ has been already deployed with OpenFaaS:

1. a client makes a HTTP `GET` (or `POST` if it needs to pass a payload) to the endpoint of the function that will be like `http://host:18080/function/`$f$;
2. the API handler[6] wraps the request in a memory structure with all its metadata and calls the function `scheduler.Schedule()` that in the `schedule` package is mapped to the `Schedule()` function of the currently instantiated scheduler;
3. since that call is blocking, the API handler will not return until the job has been executed;
4. upon the `Schedule()` return we will have in the same memory structure that we passed to the scheduler the job output that will be returned to the user as a HTTP response.

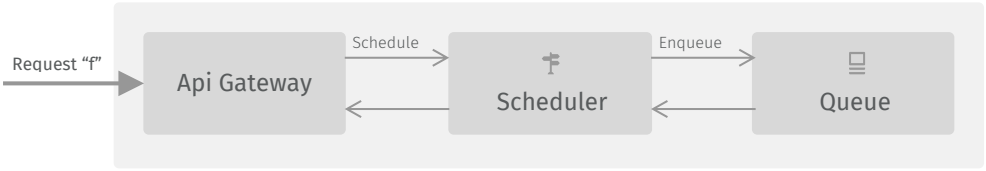


**Figure 4.4.** The processing of a function execution request in P2PFaaS

In next sections, we will see the implementation of two schedulers, the first is not exactly a scheduler, but a handler that executes functions only in the current machine, the second is an implementation of a scheduler defined in literature.

[6] i.e. the function that handles the `function<name-of-function>` API endpoint

The system is conceived in such a way that an implementation of a scheduler should never run the function directly but it should always add it to the queue that then executes it, so every scheduler that will be presented here will perform the enqueue operation when the job needs to be executed locally.

### Implementation of a No-Scheduler

This first kind of scheduler is a simple starting example and it is used for benchmarking a single machine without including the cooperation with other nodes. The only condition that it is possible to change is deciding whether to use the queue or not, in literature when the queue is not used we have a **loss model**. When loss condition is enabled, and a job request execution is received, if we have at least one free slot (that means that the number of currently running function is less than the maximum number of running functions) the job is executed otherwise it is rejected.

The flow of its operations is the following:

1. receive a job execution request (defined by the structure `ServiceRequest`)
2. check if loss condition is enabled and if we have enough slots
3. add the job to the queue

**Algorithm 4.2** NoScheduler. A basic implementation of a scheduler that only executes jobs in the current machine.

```
1: procedure SCHEDULE(j)
2:     freeSlots ← memdb.GetFreeSlots()
3:     if loss and freeSlots < 0 then return
4:     queue.EnqueueJob(j)
```

### Implementation of Power-Of-N algorithm

The Power-Of-N scheduling is based on the following consideration: if the machine that received the function execution request has that its load exceeds a certain threshold, then it asks the load level to N random nodes and it sends the request to the less loaded one, but if no other node is less loaded than it than the function is executed locally. This idea of scheduling is effectively a cooperative load balancing paradigm.

The scheduler that we are going to implement must follow the previous consideration and should have the following tunable parameters:

- $F$ is the Fanout, that is the number of nodes that will be probed for their load level and among this eventually it will be chosen the one that will execute the function;
- $T$ is the threshold, that is the level of load according to which it's needed to start asking to other nodes; in particular, the system should start cooperating when the $(K - T)$th function execution request arrives;

- *MaxHops* that is the maximum number of nodes that a job may pass through before being executed (included the node that executes the job), this is only a generalization and in our tests, it will be always set to 1;
- *Loss* that tells if the queue must be used or not; if it is true, then when the $(K+1)$th function execution request arrives it will be discarded.

**Algorithm 4.3** Power-Of-N. In the Power-Of-N scheduler the cooperation is enabled only when the load of the current machine exceeds $K-T$. In the following algorithm we assume that: RunningFunctions is the number of function that are currently running in the machine, RunningFunctionsMax is $K$ namely the maximum number of function that can run in parallel in the machine, j.PeersList is the list of peers that handled the job and j.External tells if the job is coming from another fog node.

```
1: procedure SCHEDULE(j, F, T, MaxHops, Loss)
2:     balancing ← RunningFunctions ≥ RunningFunctionsMax − T
3:     mustExecute ← j.External and j.PeersList ≥ MaxHops
4:     if balancing and not mustExecute then
5:         lessLoaded ← GetLessLoadedOfNRandom(F)
6:         if lessLoaded.RunningFunctions < RunningFunctions then
7:             ExecuteJobExternally(j, lessLoaded)
8:             return
9:     freeSlots ← memdb.GetFreeSlots()
10:    if Loss and freeSlots < 0 then return
11:    queue.EnqueueJob(j)
```

With these considerations it's possible to define a sketch of the steps that the scheduler follows (we call job a function execution request):

1. a new job execution request is **scheduled** by the API Gateway by calling the `Schedule()` function;
2. the scheduler **checks** that two conditions are verified at the same time: the balancing condition is reached, meaning that the current number of running function exceeds the threshold, and that the request reached the last node of its path and must be executed here. The only case in which the cooperation can be started is when the first condition is true and the second is not, namely in the case in which the load balancing condition is not achieved or the function must necessarily be executed here since it reached the maximum number of hops that it can do;
3. now if the previous condition is not verified then the job is **enqueued** in the current machine, otherwise, from the `discovery` service $F$ **random machines** are obtained and, among these machines, only the less loaded is picked;
4. the scheduler now **checks** if the less loaded machine is also less loaded than the current one and if this is the case then the job is sent to that machine, otherwise it is executed locally.

Now that we know the overall idea of the scheduler operations we can go a little in deep about the effective implementation of this scheme. In particular, when a job is sent from a machine to another one we will have a chain of waitings that should be closed when the last machine, which received the job to run, has effectively executed the job. The trickiest part of the implementation consists of updating a memory structure that is filled upon the job completion, in other words: the client sends a function execution request to the first node, suppose node (A), then this node forwards the job to (B) that forwards it (C) so the client waits (A) that waits (B) that waits (C). The system is implemented in such a way that every node when forwards the job to the other adds to the job object (that is then JSON serialized before sending) an item in the `PeersList` field (see Listing 4.5) representing the current node. When the last node (C) executes the job its response to (B) will have the full list of peers and the job output, so (B), when it receives it, will update find its entry in the peers' list, and it will add its timing information. The first node (A), which received the client request, will instead reply to the client with the HTTP payload represented by the job output and as HTTP headers all the metadata that the other nodes added (see ), as:

- list of IPs and IDs of peers that handled the job;
- number of hops
- a list of all the timings information (see Appendix A.1);

```
type PeerJobRequest struct {
  PeersList   []PeersListMember `json:"peers_list"`
  Function    faas.Function     `json:"function"`
  Payload     []byte            `json:"payload"`
  ContentType string            `json:"content_type"`
}

type PeerJobResponse struct {
  PeersList  []PeersListMember `json:"peers_list"`
  Body       string            `json:"body"`
  StatusCode int               `json:"status_code"`
}

type PeersListMember struct {
  MachineId string  `json:"machine_id"`
  MachineIp string  `json:"machine_ip"`
  Timings   Timings `json:"timings"`
}

type Timings struct {
  ExecutionTime     float64 `json:"execution_time"`
  FaasExecutionTime float64 `json:"faas_execution_time"`
  QueueTime         float64 `json:"queue_time"`
  ForwardingTime    float64 `json:"forwarding_time"`
  ProbingTime       float64 `json:"probing_time"`
```

```
}
```

**Listing 4.5.** P2PFaaS scheduler main types. The main struct types that are used in the scheduler service are `PeerJobRequest` that represents a function execution request exchanged between nodes, `PeerJobResponse` that is the response to the request, `PeersListMember` that represent a node that executed handled the job, `Timings` that is a set of timings used (see Appendix A.1).

## 4.4 Discovery

The discovery service plays an important role in a peer-to-peer scheduling mechanism since machines need to know to which ask for executing a job.

The discovery service that comes bundled with P2PFaaSis a very basic implementation of health-check-based polling mechanism. The discovery service follows the requirements:

1. it allows the machine to know which are the other nodes that are currently alive;
2. it automatically polls all the current known nodes to check if they are still alive;
3. if a node is not passing the alive check then is marked as dead and removed from the list of known nodes.

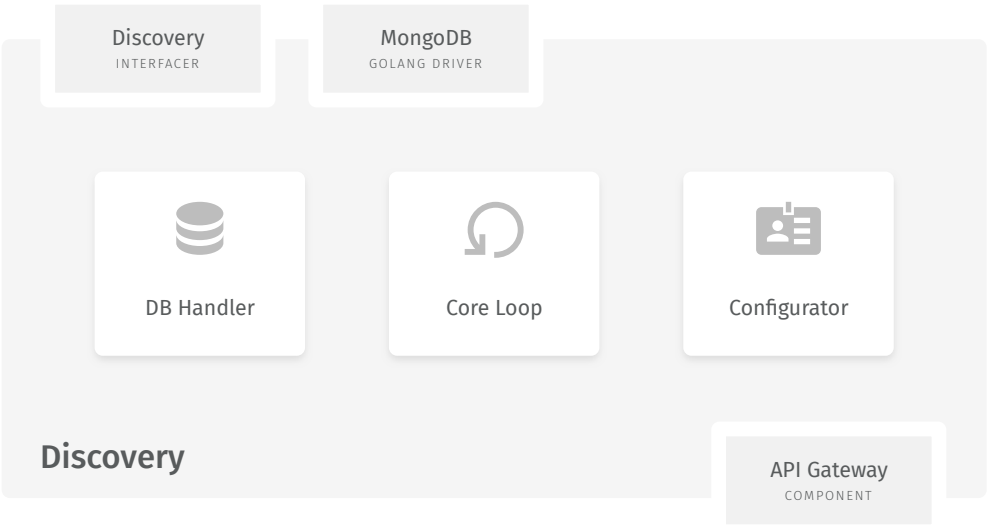


**Figure 4.5.** The P2PFaaS's discovery architecture. Discovery is composed of different units that are represented by Go packages.

The overall idea is the following. Every node has a database where all the known

machines are stored and it runs an endless loop that every pre-defined amount of time loops over the known nodes asking them the list of their known nodes. In this way, the current node can: update its list of known nodes and check if the polled nodes are alive. How to start this process? Every node has a set of init machines IP that they will probe at the boot of the service. Obviously, every set must have at least an element in common with other nodes otherwise we will have nodes that never know each other.

The discovery service architecture is depicted in Figure 4.5. The service uses two components that allow the service to interface with other applications or other nodes:

- **discovery interfacer** that allows the communication between the discovery service of another node;
- **mongodb driver** which allows the communication with the MongoDB database that is running in the current node.

Then it has three inner components:

1. **db handler** that manages the interaction with the database;
2. **core loop** that polls other machines continuously and updates the current list of known machines;
3. **configurator** that manages the current configuration of the service.

As a final component, the discovery service uses an **API Gateway** to make its features available to scheduler service.

### 4.4.1 API Gateway

The main endpoints available from the API Gateway are:

1. `/list` that is available only via `GET` method and it has a double-sided operation: the first is to return the list of available nodes (with other metadata like the ping time to the node, the id and the IP), the second is to update or add the node that called the API to the current list of available nodes (this is done only if the User-Agent of the HTTP call is the string "`Machine`");
2. `/configuration` if called via `GET` it returns all the configuration values of the parameters, otherwise with a `POST` it allows to update the runtime version configuration and the configuration file that is automatically loaded at boot.

No other endpoint is requested.

### 4.4.2 DB Handler

The main type that represents a machine is presented in Listing 4.6. The minimal set of fields that are needed to implement a service like the one that we have in mind are:

- the IP of the machine;
- the name of the machine, that is like the Linux hostname;
- the name of the fog net, if the node belongs to a specific network of fog nodes;
- the ping time;
- if the machine is declared alive or not;
- the number of times the machine did not reply to poll;

```
type Machine struct {
  IP          string
  Name        string
  FogNetName  string
  Ping        float64
  LastUpdate  time.Time
  Alive       bool
  DeadPolls   uint
}
```

**Listing 4.6.** P2PFaaS discovery machine struct type.

### 4.4.3 Configurator

The configurator component is designed as it happened in the scheduler (Section 4.3.2) and again the configuration is read from a file at boot. The main parameters are:

- listening port, that describes itself
- machine IP and machine id, that are needed metadata since the IPs conventionally is not retrieved automatically from the polling HTTP request (the reason behind this is explained in Section 4.5.1);
- **init servers**, a hard-coded list of known servers for allowing the discovery service to boot up;
- **poll time**, that is the interval between the polling of all the known nodes;
- **poll timeout**, that is the maximum time of waiting before giving up the poll operation towards a node;
- MongoDB parameters as host, user, password;
- **dead poll threshold**, that is the number of time a machine must not reply to poll before being marked as dead;
- network init **interface name**, that is needed when the IP is not explicitly defined in configuration file, in this way, at boot, the node will try to retrieve the IP from that network interface.

As it happened for the scheduler, the configuration can be altered with a HTTP `POST` to the proper endpoint.

### 4.4.4 Loop

The loop component is the core of the discovery service and it is presented in Listing 4.7.

```
func PollingLooper() {
  for ; ; {
    // step 1
    machinesToPoll, err := db.MachinesGetAliveAndSuspected()
    if err != nil {
      time.Sleep(30 * time.Second)
      continue
    }

    // step 2
    for _, m := range machinesToPoll {
      ping, err := pollMachine(m.IP)
      // check if poll succeed or not
      if err != nil {
        db.DeclarePollFailed(&m)
      } else {
        db.DeclarePollSucceeded(&m, ping.Seconds())
      }
    }

    time.Sleep(time.Duration(config.Configuration.GetPollTime()) *
    ↪  time.Second)
  }
}
```

**Listing 4.7.** P2PFaaS discovery main loop. This loop is the core of the discovery service since it keeps updated the list of known nodes by polling them at a specified interval of time.

The `PollingLooper()` function is started in a separate thread at the boot of the discovery service. Its flow of operations is the following:

1. retrieve from the database all the machines that are alive and the number of `DeadPolls` is greater than zero but less than the `DeadPollsThreshold` (so they are suspected to be down);
2. for every retrieved machine execute a poll request and, according to the response of this poll request, update the state of the machine.

## 4.5 Deploy

The two services of P2PFaaS framework are containerized using Docker. This process happens thanks to the service deployment that is made possible by Docker when a node is belonging to a swarm.

The effective deployment of the system has been done both in a local machine, by mean of the `docker-machine` utility and in a VMware ESXi server by manually instantiated virtual machines.

### 4.5.1 Docker services setup

As already described in Section 4.1.3 we can exploit the service concept of Docker Swarm to deploy the P2PFaaS's services. But before using services we need the Docker images, therefore in next sections, we will see how scheduler and discovery images are built and finally how these are declared as services and composed as a single stack.

#### Scheduler & Discovery

The scheduler and discovery images are built by using the `Dockerfile` shown in Listing 2.1. As you will remember the `Dockerfile` is a declarative file in which we can declare Docker images.

```
# => image #1
FROM golang:1.12.1-alpine3.9 as build
LABEL stage=builder

RUN apk update && apk add curl git

# install dep
RUN curl https://raw.githubusercontent.com/golang/dep/master/install.sh | sh

WORKDIR /go
COPY . .

# install deps and build
RUN cd src/scheduler && dep ensure
RUN go build scheduler

# => image #2
FROM alpine:3.8

WORKDIR /home/app
COPY --from=build /go/scheduler .

EXPOSE 18080

CMD ["./scheduler"]
```

**Listing 4.8.** P2PFaaS scheduler `Dockerfile`. This `Dockerfile` actually uses two images, the first for building the scheduler, and the second for running it.

Since Go is a compiled language the strategy of creating the executable consists of two steps:

1. the actual **compiling** of the code by using a golang builder image;
2. the **running** of the executable in a final image.

This division of the process is due to the fact that building images (as for example the one for Go) are bigger in size with respect a simple Alpine Linux[7] images used only for running the server.

In a `Dockerfile`, images are defined by the directive `FROM`. In particular, both for the scheduler and the discovery we use:

1. `golang:1.12.1-alpine3.9` that is an image based on alpine and ready for compiling Go projects but it is also enriched with `dep` that is a well-known packet manager for Go;
2. `alpine:3.8` that is a clean alpine Linux image used only for running the scheduler/discovery executable.

The `Dockerfile` used for the scheduler and discovery actually the same and what changes is only the name of the executable and of directories. The images are built with the commands:

- `docker build -t scheduler`
- `docker build -t discovery`

Then the images can be used in service definition, as explained in next section.

### Composition

Service definition and composition are done with a `docker-compose.yml` file, a configuration file in which it's possible to define services (that are based on docker images) and compose them building what, in a Docker Swarm context, is called stack.

```
version: "3.3"
services:
  scheduler:
    image: scheduler
    volumes:
      - "/var/lib/boot2docker:/var/lib/boot2docker"
    environment:
      env: production
```

[7] Alpine Linux [1] is a very lightweight Linux distribution that is often used for DevOps due to its size and flexibility.

```
      P2P_FOG_DEV_ENV: production
    ports:
      - 18080:18080
    networks:
      - func_functions
    secrets:
      - basic-auth-user
      - basic-auth-password
    deploy:
      restart_policy:
        condition: any
        delay: 5s
```

**Listing 4.9.** P2PFaaS scheduler service definition in `docker-compose.yml`.

In Listing 4.9 the definition of the scheduler service is presented, and it is composed of:

1. the **image** to use, whose name is the same built in Section 4.5.1
2. the **environment** variables, that in this case are used for setting the production environment;
3. the **ports** that are used;
4. the Docker **network** to which the container will be attached, that is `func_functions` a Docker network that is created by OpenFaaS, and we need to attach to this in order to use the OpenFaaS API service;
5. the **secrets**, that are the ones declared by OpenFaaS and they are needed to exploit the OpenFaaS API facilities;
6. **deploy** conditions, like the auto-restart of the container in case of crash;

Elements (4) and (5) are declared as "external" in Listing 4.10 since they are created by OpenFaaS when it is deployed.

```
networks:
  func_functions:
    external: true
secrets:
  basic-auth-user:
    external: true
  basic-auth-password:
    external: true
```

**Listing 4.10.** P2PFaaS services secrets and network definition in `docker-compose.yml`.

Another service that is declared in P2PFaaS's stack is MongoDB that is used by the discovery service. In Listing 4.11 its definition is presented, but now the image `mongo:latest` is pulled from the official Docker image repository.

```
mongo:
image: mongo:latest
restart: always
environment:
  MONGO_INITDB_ROOT_USERNAME: root
  MONGO_INITDB_ROOT_PASSWORD: example
networks:
  - func_functions
```

**Listing 4.11.** P2PFaaS services MongoDB definition in `docker-compose.yml`.

The final P2PFaaS `docker-compose.yml` file is composed, in order, by Listing 4.9 for scheduler and discovery, then 4.11 and finally 4.10. Once all the images are built the stack can be deployed with the command `docker stack deploy`, in particular, a bash script that allows to completely deploy the stack is shown in Listing 4.12 and it is useful when the same stack must be deployed in a high number of different machines.

```
#!/bin/sh
# deploy the stack
cd ../../
cd stack-discovery
docker build -t discovery:latest .
cd ..
cd stack-scheduler
docker build -t scheduler:latest .
cd ..
cd stack
docker stack rm p2p-fog
docker stack deploy -c docker-compose-local.yml p2p-fog

# remove unused images and containers
docker system prune -f --volumes
```

**Listing 4.12.** P2PFaaS `bash` deploy script. This script allows the complete deploy of the framework, and it is run with the full clone of all the `git` repositories of P2PFaaS. Note that the last command is used for cleaning all the data used by the old deploy of the framework since before deploying a new version the old one is dismissed with the command `docker stack rm p2p-fog`.

Figure 4.6 shows the output of the command `docker service ls`, that is used to show a complete list of all the deployed services in a swarm, after a full deploy of P2PFaaS , OpenFaaS and the *Pigo Face Detector* function (see Chapter 5).

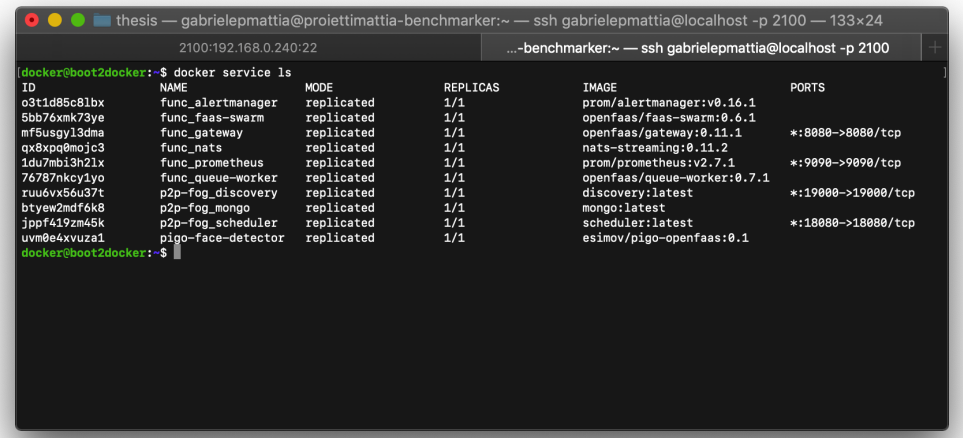


**Figure 4.6.** Docker Swarm services after a full deploy of P2PFaaS

### 4.5.2 Testing environment with `docker-machine`


**Figure 4.7.** `boot2docker` virtual machine starting up

Docker Machine [17] is a tool, officially provided by Docker, which allows an easy an fast configuration of virtual machine with Docker installed. A very reductive list of Docker Machine's features comprehends the creation of virtual machines and their management via terminal without ssh but using the Docker API facilities.

The first P2PFaaS's test-bed was deployed by creating a number of virtual machines with the command:

```
docker-machine create -d virtualbox <vm-name>
```

And then its management can be enabled with the command:

```
docker-machine env <vm-name>
```

Specifically, this allows to use all the Docker commands in the terminal as we were on the specified machine, not by using ssh but exploiting the Docker APIs that are exposed at Docker daemon's port `2375`.

#### `boot2docker` and VM configuration

The standard VM image that is used by `docker-machine` is `boot2docker` [10] (Figure 4.7), a very lightweight Linux image that is specifically designed for running Docker. Actually, the image is a fork of TinyCore Linux [39].

The particularity of this image is that it runs completely on RAM but it allows you to attach also a disk in order to make some changes persistent. These changes regard, for example, the deployed services that must be run at the machine boot, the ssh keys and other tools that are needed. In this environment, no other service is provided beyond Docker, this also allows to have a clean space where Docker services run without possible noise of unwanted services.

Since `docker-machine` allows complete management of the VM from the host terminal, no particular configuration of the vm image is done, since the P2PFaaS's source code is loaded from the host machine and all the commands are issued from the host.

### 4.5.3 Production environment with *VMware ESXi*

The production environment in which P2PFaaS has been deployed is the one in which benchmarks have been done (Chapter 5). In this environment I decided not to use `boot2docker` image since its usage in production is highly discouraged [10].

The base Linux image that I used is based on Debian 9 "Stretch" [15] that I prepared in the following way:

1. installed Docker;
2. installed P2PFaaS node ssh key and authorized benchmarker's one;
3. installed OpenFaas by pulling from GitHub;
4. installed P2PFaaS by pulling all the relative repositories

This image has been then cloned for having in total 8 machines.

### 4.5.4 Production environment with *Raspberry Pi*

Deploying the framework in *Raspberry Pi* [35], the well-known low cost board with Linux, required me to use another base image since I decided not to use the default Raspbian[8] [36] image which does not come with Docker. Indeed, the installation of Docker in the Raspberry Pi requires to use a script which adds new repositories to `apt`. For this reason, I deployed the framework on a base image called "HypriotOS" that is an ARM-compatible image designed to work with Raspberry Pi that also comes with Docker pre-installed. The project is mature and the image is stable and it is perfect for deploying OpenFaaS and the P2PFaaS framework.

Starting from this image the followed the same preparing process that I described for Debian (Section 4.5.3).

[8] As the name suggests, it is a fork of Debian that is heavily optimized for running in Raspberry Pi.

# Chapter 5

# Benchmarks

By having defined the theoretic and the implementation part of the framework we can now analyze its performance. Indeed in the next sections, I will present a series of plots of benchmarks make also a comparison with the mathematical model. The plots will deal with:

- **single machine** benchmarks, to prove that effectively the *M/M/1/K-PS* model fits the framework implementation when cooperation is disabled;
- **long run** benchmarks, to show the advantage offered by the *LL-PS(T)* algorithm with respect to the non-cooperating model;
- **best threshold** benchmarks, to understand which should be the best threshold to use for the *LL-PS(T)* algorithm;
- **resources** charts, to inspect the impact of the algorithm on CPU, RAM and network resources.

The benchmarks that I will present will regard the *Pigo Face Detector* function, a prebuilt function which is able to detect faces within a photograph. For using this function the image must be passed as HTTP payload and the response will be the same image but with yellow squared faces. Therefore a certain part of the network transmission delay comprehends the exchange of the photo.

## 5.1 Environment

Results of benchmarks that are presented in the next sections have obtained done by using a total of three servers with VMWare ESXi as bare metal hypervisor. Every server is an IBM BladeCenter HS22 with two Intel Xeon X5560 @ 2.80Ghz for a total of 8 physical and 16 logical cores, and 25GB of RAM. These three servers have been configured in this way:

- Server #1 has 4 VM with Debian 9 and P2PFaaS deployed, every VM has 1 core, 3GB of RAM and 8GB of disk assigned and reserved; in this setup, every VM represent a fog node;
- Server #2 is as Server #1;

- Server #3 has 1 VM with CentOS, 8 cores, 4GB of RAM and 25GB of disk; this VM is the one that performs benchmarks.

So benchmarks of the framework with *LL-PS(T)* have been conducted on a total of 8 physical nodes. Benchmarks on single nodes have been conducted in the same environment but with cooperation disabled.

In this testing environment, the *Pigo Face Detector* has a duration of $\approx 0.27s$ and $\mu_f = 3.70$, I also assume (if not explicitly stated) that the maximum number of parallel running jobs is $K = 10$.

## 5.2 Single Machine

The first kind of tests that I conducted on P2PFaaS was related to measuring if a single machine with cooperation disabled reflected the *M/M/1/K* model. As we already discussed in Section 2.1.3, if we consider the delay as seen by the client and the blocking probability, models *M/M/1/K* and *M/M/1/K-PS* are equivalent.

Since OpenFaaS does not put any limit on the number of functions that are running in parallel on a swarm I used my framework with the *NoScheduler* scheduler that we discussed in Section 4.3.5. In this way the only feature that is used from the framework is dropping requests when load is $K$.

Figures 5.1 and 5.2 shows the *Pigo Face Detector* function with $K = 10$ and $K = 20$ respectively compared with the *M/M/1/K* model. As we can easily see, the experiments confirms that the model is correct, namely a single physical machine that drops jobs when load is $K$ and executes jobs in parallel is modeled with a *M/M/1/K*. This has been foreseen since models *M/M/1/K-PS* and *M/M/1/K* are equivalent if we only consider the average total delay $W$ for executing a job and the blocking probability $P_B$.

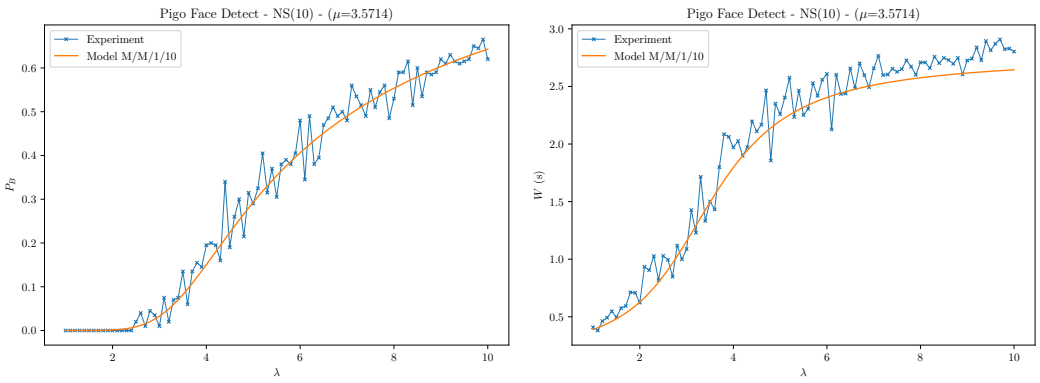


**Figure 5.1.** $P_B$ and $W$ charts for without cooperation for *Pigo Face Detect* and $K$ set to 10 compared with the *M/M/1/K* model

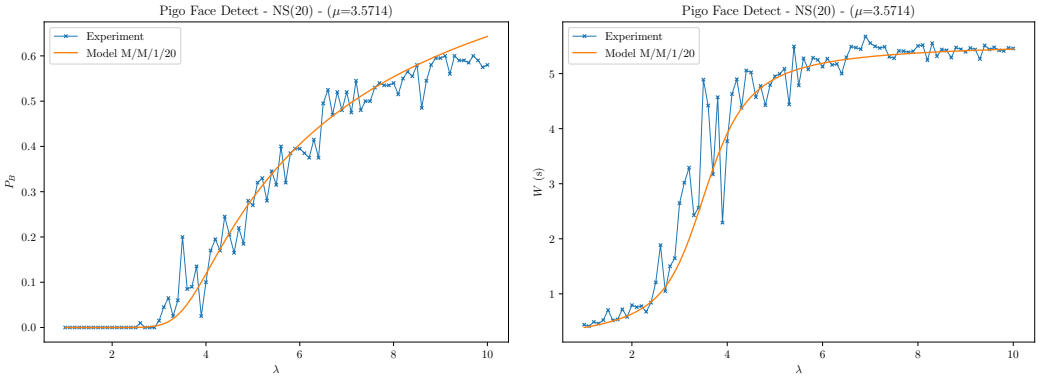


**Figure 5.2.** $P_B$ and $W$ charts without cooperation for *Pigo Face Detect* and $K$ set to 20 compared with the *M/M/1/K* model

## 5.3 Beneficial Effect of Cooperation

In this section, I will present a series of benchmark that has been done on the *LL-PS(T)* algorithm to measure its convenience compared with the case in which the cooperation is not used. I called them "Long Run" tests because they have been done on a wide range of $\lambda$ but with a low number of requests in order to make them last about 24 hours.

### 5.3.1 *LL-PS(K-1)*

The first set of benchmarks of the *LL-PS(T)* algorithm has been done by choosing and $T = K - 1$, the reason of this particular value will be explained later, now we only focus on the convenience that the cooperation introduces with respect to the case in which a machine only executes jobs on its own.

In order to reach a good compromise between the duration of the test and the "stability"[1] of the results I chose a range of $\lambda$ from 2.0 to 3.6, since $\mu = 1/0.27$ then it makes no sense to go beyond $\lambda = 3.6$ given that for having $\rho = 1$, that is the saturation point, then $\lambda = 3.7$.

Figure 5.3 shows the results of the benchmark. What clearly emerges is that the cooperation allows to reduce the blocking probability, in other words, it allows to carry out more jobs with respect the non-cooperation case, that is represented by the model *M/M/1/10-PS*. This because when we start to do probing then it is more likely that we find a machine that is less loaded than us and by forwarding to it function invocations then we are able to drop fewer requests, thus reducing the blocking probability. From the delay point of view, we can observe a different behavior, meaning that the total delay as seen by the clients is more or less equal to the case in which the cooperation is not used but until $\lambda = 3.0$ ($\rho = 0.85$) then

[1]For "stability" here I intend how much the curves oscillates, indeed if the number of samples is too low then it may be difficult to understand the results. However, the higher is the number of samples the longest is the duration of the benchmark. In these benchmarks samples are the function invocations.

the delay becomes higher. However, this is acceptable if consider the fact that more jobs are served.

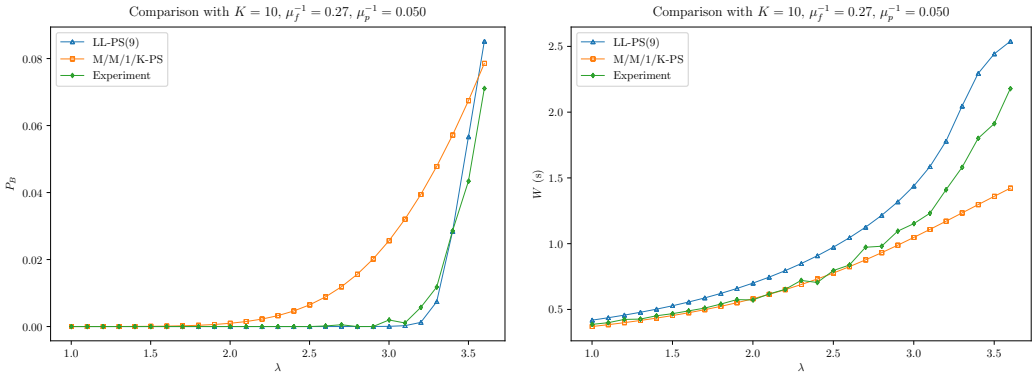


**Figure 5.3.** $P_B$ and $W$ charts of *LL-PS(K-2)* benchmark with $\lambda = [2.0, 3, 6]$ at steps of 0.1 also compared with the model described in Section 3.2.

Another characteristic that we can observe is how the model essentially catches the behavior of the system.

## 5.4 Best threshold

Since the penalty that characterizes the model *LL-PS(T)* increases with the load $\lambda$ and the threshold $T$, makes sense to understand which is the best threshold that should be used for maximizing performances. The series of benchmarks that I will present here has been focused on a single value of $\lambda$, i.e. the one that correspond to $\rho \approx 0.95$ but trying all possible values of the threshold. Again this kind of set up has been done to make the test duration of about 24hrs.

### 5.4.1 *LL-PS(T)*

Figure 5.4 shows blocking probability and $P_B$ and total delay $W$ when $\lambda$ is fixed at 3.50. The model again catches the essential behavior of the real system, in particular for the blocking probability the local minimum when $T = 9$ emerges. This point is the one that represents the best tradeoff between the beneficial effect of cooperation for load balancing and the penalty.

As far as regards the delay, the decreasing behavior is justified by the fact that when the threshold is higher the cooperation is limited. By limiting the cooperation, we also limit all the penalties that are associated with it and that makes the node to persist at a higher load. As a natural consequence, if the penalties are reduced then reduced is also the average time that jobs spend in the system.

Another key point that emerges from these charts is that the model does not exactly catch the experiments as happens when $\lambda$ varies. This can be explained if we consider the fact that the model offers an asymptotic approximation that is valid for a number of nodes $N \to \infty$. Since the experiment involved only 8 nodes such discrepancy is expected, and what is relevant is that the model catches the implications of the penalty on the performances.

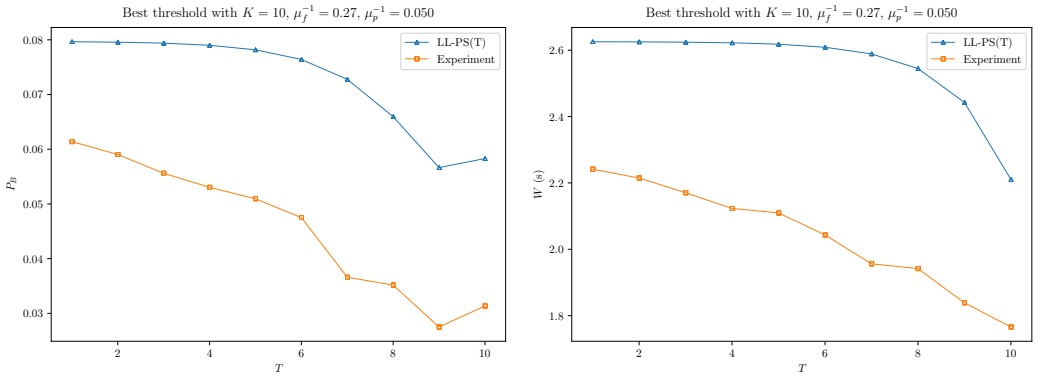


**Figure 5.4.** Best threshold chart of *LL-PS(T)* with 8 machines for the blocking probability $P_B$ and delay $W$, also compared to the model. $\lambda$ fixed at 3.50

### 5.4.2 Resources

Figures 5.5, 5.6 and 5.7 shows the resources consumption of a single node during the just discussed "Best Threshold" benchmark. The test started at 16:30 on 06/14/2019 and ended at 16:00 on 06/15/2019, then the benchmark for every value of the threshold, from 0 to 10, lasted 2 hours. The gaps that are observable in the CPU and in the network utilization charts are the 5 minutes intervals between the tests[2].

If the CPU and the RAM charts show a relatively stationary behavior, the network activity one is characterized by a depression on the first values of the threshold, $T = 10$ and $T = 9$. This because when the threshold is very high, the number of probes and job forwarding is lower and less network resource is used. When the threshold decreases, the number of cooperation interactions starts to increase, consuming network resources. Indeed, the chart shows that the network cap is reached from $T = 8$, when about 6 Mbit/s are used both in the uplink and in the downlink. If we consider that the eight benchmarking machines are connected to the same 100 Mbit/s switch, every machine's link should have a capacity $100/8 = 12.5$ Mbit/s, as the chart shows.

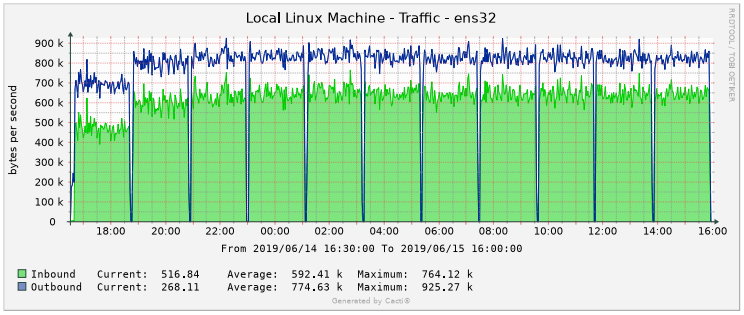


**Figure 5.5.** Network activity chart during the "Best Threshold" benchmark of *LL-PS(T)*

[2]Therefore, in total, there are 11 segments, one for every value of the threshold for $T = 10$ to $T = 0$, in order.

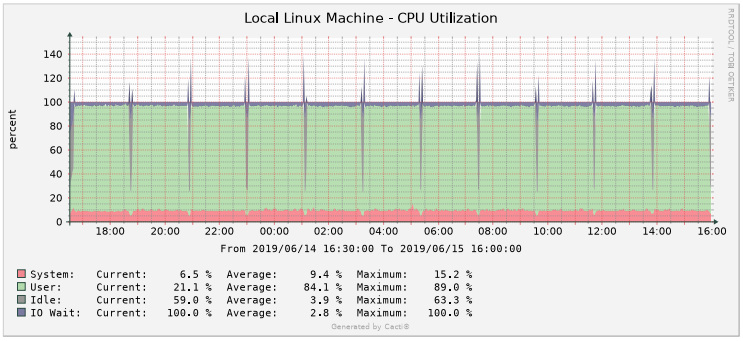


**Figure 5.6.** CPU utilization chart during the "Best Threshold" benchmark of *LL-PS(T)*

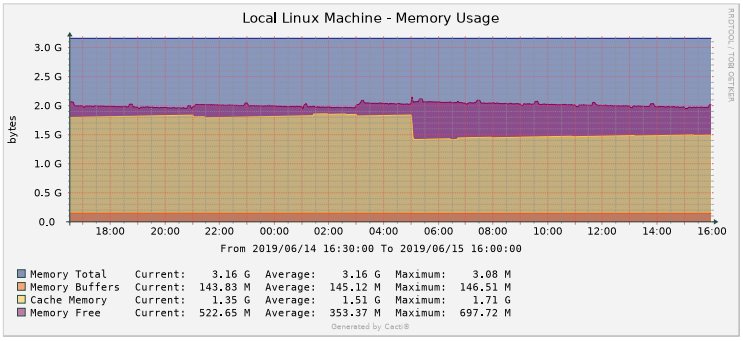


**Figure 5.7.** RAM utilization chart during the "Best Threshold" benchmark of *LL-PS(T)*

## 5.5 Timings Decomposition

When running benchmarks of a distributed algorithm, one key analysis that can be done, in order to understand how the system behaves, is the study of timings and delays. For this reason in P2PFaaS when a function invocation returns, a set of timings is returned, you can read the Appendix A.1 for a full description of them.

### 5.5.1 *LL-PS(K-1)*

For this set of charts, we again set $T = K - 1$. Figure 5.8 shows the trend of both the Forwarding and the Probing Time and since these two activities requires CPU time and network resources, it's clear that the higher the load the higher the delay since more machines are simultaneously doing the same operations and the CPU is also used for executing the functions. This reasoning explains why the probing starts from $6ms$ with $\lambda = 2.0$ and ends with $20ms$ at $\lambda = 3.60$, the same for the Forwarding Time that starts with $15ms$ when $\lambda = 2.0$ and ends with $45ms$ when $\lambda = 3.60$. When the *Pigo Face Detection* function is called, the photo to be processed is passed as a payload, and this also happens when nodes forward jobs to other nodes: the entire payload is transferred for every forwarded job. Moreover, making a probe and forwarding jobs requires that data is encapsulated and serialized in JSON. Therefore, as modeled in 3.2, probing and forwarding introduce penalties, both for the network and for CPU resources.

By having collected all the timings information we can depict a decomposition

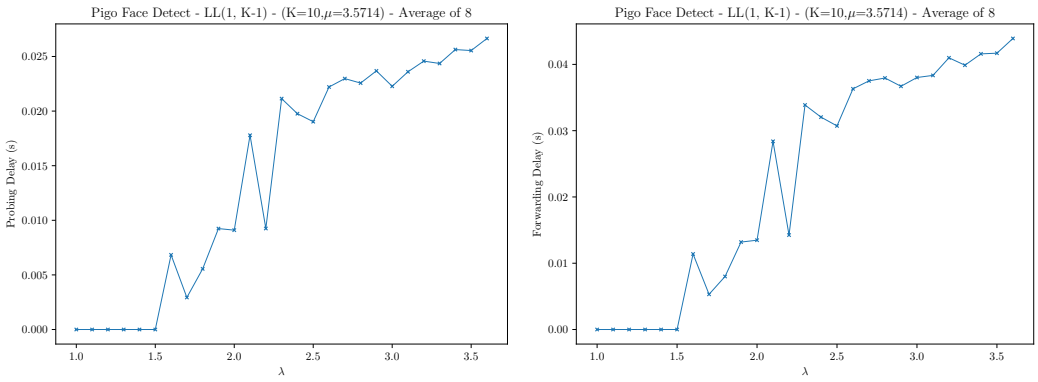


**Figure 5.8.** Average of Probing and Forwarding times for *LL-PS(K-2)* with 8 machines and $K = 10$

like the one in Figure 5.9.

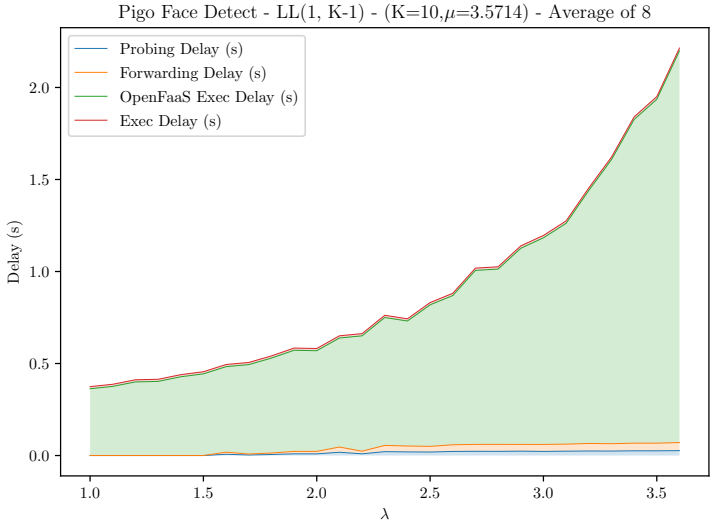


**Figure 5.9.** Average timings breakdown of *LL-PS(K-2)* with 8 machines $K = 10$.

# Chapter 6

# Conclusions

In this thesis, I tried to go through the entire path from the theory to the practical application of a cooperative load balancing algorithm. In the beginning, I proposed the idea and I tried to define the best mathematical model that is able to describe it. Then I realized it by creating an entire framework that is able to implement any possible scheduler by using FaaS as job model. As a final step, I demonstrated that what I supposed, in theory, is then verified to happen in reality.

As happens with people, cooperation between nodes is an important key factor that can be exploited in order to improve performances.

## 6.1 Future Work

I want to conclude this thesis with a glance at what are the possible improvements that can be done with the P2PFaaS framework in some aspects that I did not consider deliberately.

### 6.1.1 Security

In the development of the framework, security has not been addressed however if the system needs to be deployed in a production environment there are some improvements that need to be taken into account.

OpenFaaS implements a **basic authentication** that is done via HTTP headers but P2PFaaS does not provide any kind of authentication, this means that theoretically OpenFaaS can be managed through P2PFaaS. Therefore a first security improvement that can be done is related to the authentication of the API endpoint through HTTP headers, moreover P2PFaaS could use the same credentials that are used for OpenFaaS.

As far as regards the communication that happens between nodes, not only it is not authenticated but it is also not **encrypted**. Therefore, as a second security improvement, a sort of HTTPS connection should be implemented for allowing secure communication between nodes.

### 6.1.2 Metrics

Another aspect that could be addressed regards the metrics. In P2PFaaS the only metrics that are available are the ones that are provided in the HTTP headers of the function response. A possible improvement, that also follows the OpenFaaS philosophy regards the implementation of a standard interface of providing metrics, for example by using Prometheus [33] a monitoring framework. Prometheus should be deployed as a service aside the P2PFaaS Scheduler and Discovery services, then it can receive metric data from the framework. Prometheus will offer at a given port a standardized way for obtaining metrics data.

# Appendices

# Appendix A

# P2PFaaS's Scheduler peculiarities

## A.1 Timings decomposition

For having a deep understanding of all the possible delays that may affect the execution of a function, I inserted different time checkpoint during the execution of a function. This allows to decompose the total delay that is seen by the client.

The timings that are captured by P2PFaaS and that are returned in the HTTP headers, after the execution of a function are the following:

- *Execution Time* (also called `execTime`): it's the time between the OpenFaaS call to execute the function and the time in which the job output is returned;
- *OpenFaaS Execution Time* (also called `faasExecTime`): it's the duration time of the function as reported by OpenFaaS;
- *Probing Time* (also called `probingTime`): it's the total time for obtaining the list of all the machines, selecting $F$ at random and checking the load;
- *Forwarding Time* (also called `forwardingTime`): it's the time that is spent for transferring a job from the machine in which the job is requested to the machine that will actually execute it. This time is computed by starting a timer as soon as the job leaves the machine and stopping it when the job result is returned: from this time the "OpenFaaS Execution Time" is subtracted;
- *Queue Time* (also called `queueTime`): it's the time that a job spends in the queue before being executed;
- *Delay* (also called $W$): it's the total execution time as seen by the client.

# Appendix B

# Benchmarking scripts

An important role in the P2PFaaS framework is played by the benchmarking scripts that I designed for testing the framework. There are essentially two scripts that I used for generating all the benchmark results in Chapter 5:

1. a **single machine** benchmark script that is designed for generating a flow of requests for a single machine;
2. a **multi machine** benchmark script that is designed for generating parallel flows of requests for multiple machines.

All the scripts are written in Python and their output is then processed and plotted with another series of scripts that uses the `matplotlib` library for creating the charts.

The *single machine* benchmark takes as input the host, the URL of the function and a range of lambda values and then generates a flow of requests. This flows can also be Poissonian and, if it is the case, the waiting time between a request and the next one is picked from an exponential distribution of parameter $\lambda$. The result code of every request, that is `200` if the job is executed and `500` if it is rejected, is then written to a file by computing the total delay and by also parsing the HTTP response headers that contains other useful information about the execution of the function.

The *multi machine* benchmark script exploits the single machine one for benchmarking multiple machines in parallel by creating threads, every thread handles the benchmarking suite of a single machine. Upon the termination of all the threads, the script is then able to gather all the results in a single output file.

# List of Figures

# Bibliography


[1] *Alpine Linux - About*. URL: https://www.alpinelinux.org/about/ (visited on 04/25/2019).

[2] Hany Atlam, Robert Walters, and Gary Wills. "Fog Computing and the Internet of Things: A Review". In: *Big Data and Cognitive Computing* 2 (Apr. 2018). DOI: 10.3390/bdcc2020010.

[3] Ioana Baldini et al. *Serverless Computing: Current Trends and Open Problems*. Ed. by Sanjay Chaudhary, Gaurav Somani, and Rajkumar Buyya. Singapore: Springer Singapore, 2017, pp. 1–20. ISBN: 978-981-10-5026-8. DOI: 10.1007/978-981-10-5026-8. URL: https://doi.org/10.1007/978-981-10-5026-8.

[4] Ioana Baldini et al. "The Serverless Trilemma: Function Composition for Serverless Computing". In: Onward! 2017 (2017), pp. 89–103. DOI: 10.1145/3133850.3133855. URL: http://doi.acm.org/10.1145/3133850.3133855.

[5] R. Beraldi and H. Alnuweiri. "Sequential Randomization load balancing for Fog Computing". In: (Sept. 2018), pp. 1–6. ISSN: 1847-358X. DOI: 10.23919/SOFTCOM.2018.8555797.

[6] R. Beraldi, H. Alnuweiri, and A. Mtibaa. "A Power-of-Two Choices Based Algorithm for fog Computing". In: *IEEE Transactions on Cloud Computing* (2018), pp. 1–1. ISSN: 2168-7161. DOI: 10.1109/TCC.2018.2828809.

[7] R. Beraldi, A. Mtibaa, and H. Alnuweiri. "Cooperative load balancing scheme for edge computing resources". In: (May 2017), pp. 94–100. DOI: 10.1109/FMEC.2017.7946414.

[8] Roberto Beraldi and Hussein Alnuweiri. "Exploiting power-of-choices for load balancing in fog computing". In: *IEEE International Conference on Cloud Engineering, 2019* ().

[9] Flavio Bonomi et al. "Fog Computing and Its Role in the Internet of Things". In: MCC '12 (2012), pp. 13–16. DOI: 10.1145/2342509.2342513. URL: http://doi.acm.org/10.1145/2342509.2342513.

[10] *Boot2Docker - Github*. URL: https://github.com/boot2docker/boot2docker (visited on 04/26/2019).

[11] S. Borst, O. Boxma, and N. Hegde. "Sojourn times in finite-capacity processor-sharing queues". In: (Apr. 2005), pp. 53–60. DOI: 10.1109/NGI.2005.1431647.

[12] Maury Bramson, Yi Lu, and Balaji Prabhakar. "Asymptotic independence of queues under randomized load balancing". In: *Queueing Systems* 71.3 (July 2012), pp. 247–292. ISSN: 1572-9443. DOI: 10.1007/s11134-012-9311-0. URL: https://doi.org/10.1007/s11134-012-9311-0.

[13] A. Checko et al. "Cloud RAN for Mobile Networks—A Technology Overview". In: *IEEE Communications Surveys Tutorials* 17.1 (2015), pp. 405–426. ISSN: 1553-877X. DOI: 10.1109/COMST.2014.2355255.

[14] Edward G Coffman Jr, Richard R Muntz, and H Trotter. "Waiting time distributions for processor-sharing systems". In: *Journal of the ACM (JACM)* 17.1 (1970), pp. 123–130.

[15] *Debian - The Universal Operating System*. URL: https://www.debian.org (visited on 05/21/2019).

[16] *Docker - Swarm mode overview*. URL: https://docs.docker.com/engine/swarm/ (visited on 04/19/2019).

[17] *Docker Machine - Overview*. URL: https://docs.docker.com/machine/overview/ (visited on 04/26/2019).

[18] Geoffrey Fox et al. "Status of Serverless Computing and Function-as-a-Service (FaaS) in Industry and Research". In: (Aug. 2017). DOI: 10.13140/RG.2.2.15007.87206.

[19] David Freet et al. "Cloud Forensics Challenges from a Service Model Standpoint: IaaS, PaaS and SaaS". In: MEDES '15 (2015), pp. 148–155. DOI: 10.1145/2857218.2857253. URL: http://doi.acm.org/10.1145/2857218.2857253.

[20] David G. Kendall. "Stochastic Processes Occurring in the Theory of Queues and Their Analysis by the Method of Imbedded Markov Chains". In: *The Annals of Mathematical Statistics* 24 (Sept. 1953). DOI: 10.1214/aoms/1177728975.

[21] Fabrizio Granelli. *The Role of Cloud and MEC in 5G*. 2019.

[22] X. He et al. "A novel load balancing strategy of software-defined cloud/fog networking in the Internet of Vehicles". In: *China Communications* 13.Supplement2 (2016), pp. 140–149. ISSN: 1673-5447. DOI: 10.1109/CC.2016.7833468.

[23] Cheol Ho Hong and Blesson Varghese. "Resource Management in Fog/Edge Computing: A Survey". In: *CoRR* abs/1810.00305 (2018). arXiv: 1810.00305. URL: http://arxiv.org/abs/1810.00305.

[24] J. Kennedy and R. Eberhart. "Particle swarm optimization". In: 4 (Nov. 1995), 1942–1948 vol.4. DOI: 10.1109/ICNN.1995.488968.

[25] Leonard Kleinrock and Richard Gail. *Queueing systems*. Wiley, 1986.

[26] M. Mitzenmacher. "The power of two choices in randomized load balancing". In: *IEEE Transactions on Parallel and Distributed Systems* 12.10 (Oct. 2001), pp. 1094–1104. ISSN: 1045-9219. DOI: 10.1109/71.963420.

[27] Michael Mitzenmacher, Andréa W. Richa, and Ramesh Sitaraman. "The Power of Two Random Choices: A Survey of Techniques and Results". In: (2000), pp. 255–312.

[28] R. K. Naha et al. “Fog Computing: Survey of Trends, Architectures, Requirements, and Research Directions”. In: *IEEE Access* 6 (2018), pp. 47980–48009. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2866491.

[29] S. Ningning et al. “Fog computing dynamic load balancing mechanism based on graph repartitioning”. In: *China Communications* 13.3 (Mar. 2016), pp. 156–164. ISSN: 1673-5447. DOI: 10.1109/CC.2016.7445510.

[30] *OpenFaaS*. URL: https://www.openfaas.com/ (visited on 04/19/2019).

[31] *OpenFaaS - Gateway*. URL: https://docs.openfaas.com/architecture/gateway/ (visited on 04/20/2019).

[32] *OpenFaaS - Watchdog*. URL: https://docs.openfaas.com/architecture/watchdog/ (visited on 04/19/2019).

[33] *Prometheus - Monitoring system and time series database*. URL: https://prometheus.io (visited on 05/22/2019).

[34] D. Puthal et al. “Secure and Sustainable Load Balancing of Edge Data Centers in Fog Computing”. In: *IEEE Communications Magazine* 56.5 (May 2018), pp. 60–65. ISSN: 0163-6804. DOI: 10.1109/MCOM.2018.1700795.

[35] *Raspberry Pi - Teach, Learn, and Make with Raspberry Pi*. URL: https://www.raspberrypi.org (visited on 05/21/2019).

[36] *Raspbian*. URL: https://raspbian.org (visited on 05/21/2019).

[37] Mahadev Satyanarayanan, Wei Gao, and Brandon Lucia. “The Computing Landscape of the 21st Century”. In: HotMobile ’19 (2019), pp. 45–50. DOI: 10.1145/3301293.3302357. URL: http://doi.acm.org/10.1145/3301293.3302357.

[38] T. Taleb et al. “On Multi-Access Edge Computing: A Survey of the Emerging 5G Network Edge Cloud Architecture and Orchestration”. In: *IEEE Communications Surveys Tutorials* 19.3 (2017), pp. 1657–1681. ISSN: 1553-877X. DOI: 10.1109/COMST.2017.2705720.

[39] *Tiny Core Linux*. URL: http://www.tinycorelinux.net (visited on 04/26/2019).

[40] Qiaomin Xie et al. “Power of D Choices for Large-Scale Bin Packing: A Loss Model”. In: *SIGMETRICS Perform. Eval. Rev.* 43.1 (June 2015), pp. 321–334. ISSN: 0163-5999. DOI: 10.1145/2796314.2745849. URL: http://doi.acm.org/10.1145/2796314.2745849.

[41] K. Zhang et al. “Energy-Efficient Offloading for Mobile Edge Computing in 5G Heterogeneous Networks”. In: *IEEE Access* 4 (2016), pp. 5896–5907. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2016.2597169.