Architecture and Performance Evaluation of Distributed Computation Offloading in Edge Computing

Claudio Cicconetti[a,*], Marco Conti[a], Andrea Passarella[a]

[a] *IIT, National Research Council, Pisa, Italy*

Abstract

Edge computing is an emerging paradigm to enable low-latency applications, like mobile augmented reality, because it takes the computation on processing devices that are closer to the users. On the other hand, the need for highly scalable execution of stateless tasks for cloud systems is driving the definition of new technologies based on serverless computing. In this paper, we propose a novel architecture where the two converge to enable low-latency applications: this is achieved by offloading short-lived stateless tasks from the user terminals to edge nodes. Furthermore, we design a distributed algorithm that tackles the research challenge of selecting the best executor, based on real-time measurements and simple, yet effective, prediction algorithms. Finally, we describe a new performance evaluation framework specifically designed for an accurate assessment of algorithms and protocols in edge computing environments, where the nodes may have very heterogeneous networking and processing capabilities. The proposed framework relies on the use of real components on lightweight virtualization mixed with simulated computation and is well-suited to the analysis of several applications and network environments. Using our framework, we evaluate our proposed architecture and algorithms in small- and large-scale edge computing scenarios, showing that our solution achieves similar or better delay performance than a centralized solution, with far less network utilization.

Keywords: online job dispatching, serverless computing, computation offloading, edge computing, performance evaluation

1. Introduction

Edge computing[1] is generally considered the principal enabling technology for several applications with stringent latency requirements [1]. With edge com-

*Corresponding author

Email addresses: c.cicconetti@iit.cnr.it (Claudio Cicconetti), m.conti@iit.cnr.it (Marco Conti), a.passarella@iit.cnr.it (Andrea Passarella)

[1] In the scientific literature and market press the terms "fog" and "edge" are used with overlapping vs. different meaning under varied circumstances, which often depend on the

puting, processing capabilities are shifted from remote data centers to networking devices in the access network with spare or added computational capabilities, which are closer the end users [2]. Several market segments are calling for edge computing technologies, since it would enable new and important applications. This is confirmed by the huge participation of industrial parties to standardization bodies and alliances, e.g., the OpenFog Consortium [3] and European Telecommunications Standards Institute (ETSI) Multi-access Edge Computing (MEC) ETSI [4]. However, the promises of edge computing are limited in the presence of mobile devices. This is shown with the help of the example in the box "Traditional edge computing" of Fig. 1: an edge client is initially assigned resources on a given edge node that best fits its location and requirements; later on, once the mobile device roams to another part of the network, the system must either migrate the application from the original edge node to a newer one (which has an overhead and may also cause a temporary interruption of the application execution) or accept that some traffic is re-routed appropriately to the formerly closest edge node (which increases application latency).

In this paper we provide two major contributions. Firstly, we propose a new architecture for the execution of serverless tasks, distributed on edge nodes. With this approach we overcome the above limitation of traditional edge computing by removing the overhead incurred by a user when roaming. At the same time we extend the state-of-the-art on serverless computing since our solution is designed for a heterogeneous environment of devices with limited communication and computational capabilities. Secondly, we recognize that the tools for the performance evaluation of edge computing systems are quite limited, notwithstanding the high interest in this technology, hence we propose a novel framework for the performance analysis that is suitable to a wide range of conditions of practical interest. Both contributions are introduced briefly in the following.

Serverless computing [5] is an emerging concept in cloud computing that extends the paradigm of micro-services that is dominating most modern deployments towards the client applications. A micro-service is a scalable application that responds to function calls from a remote agent. Serverless computing means that a client on the end user device may request such executions, often via short-lived jobs in a scripting language, which do not have a state on the remote execution server, neither soft nor on persistent storage. Thus, no obvious choke point exists to limit performance as the number of clients or requests grow, provided that new resources can be added to upscale the micro-services. Amazon has been among the first to offer a commercial service for serverless computing, called AWS Lambda, closely followed by Microsoft Azure functions and Google Cloud functions, as well as several open source projects such as

specific context or application use case. The concepts illustrated in this paper apply to a wide range of systems, including both flavors of computing and communication systems, therefore we do not clearly define the fog/edge terms, but rather always refer to *edge* computing to avoid ambiguity.

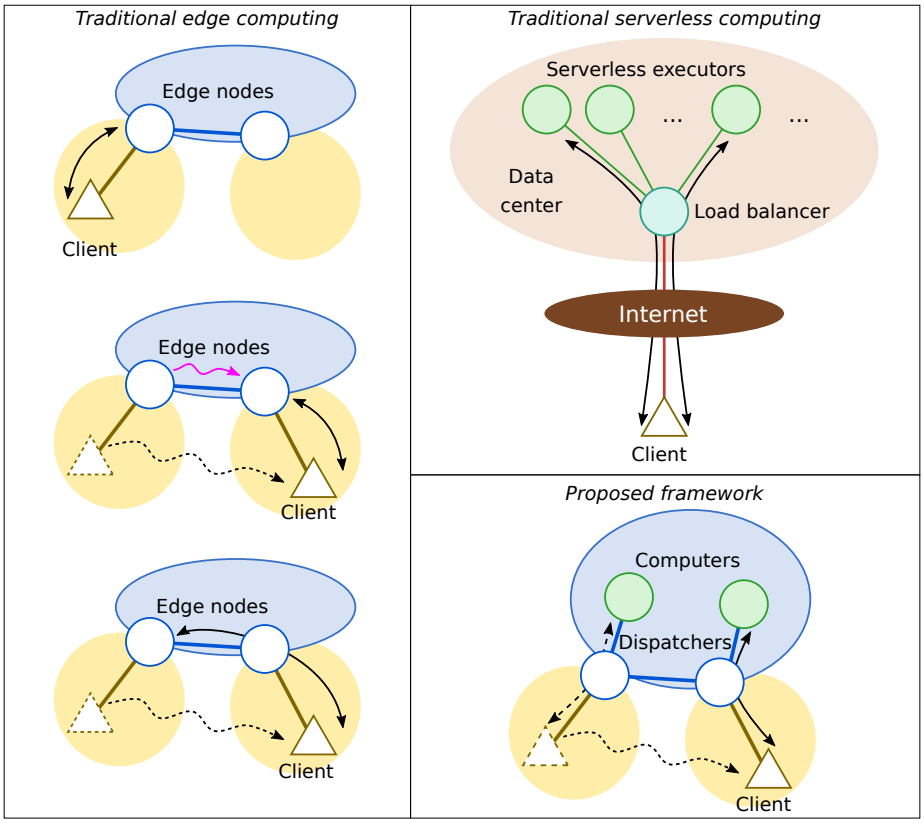

Figure 1: Comparison of traditional edge computing and serverless architectures with the proposed solution, which combines both.

Apache OpenWhisk[2] and Knative[3]. As shown in the "Traditional serverless computing" box of Fig. 1, all these solutions adopt a logically centralized approach to load balance the incoming requests to a pool of servers, which are assumed to be located in a data center. On the contrary, in edge computing the servers may have heterogeneous hardware characteristics and they are generally distributed across large areas, which introduces non-negligible communication delays, much higher than that of high-speed transfer networks in a data center. This makes the existing serverless solutions unsuitable to be deployed in an edge system.

In this paper, we propose a novel solution for the computation offloading in edge systems, based on the execution of *lambda functions* on edge nodes, as illustrated in the "Proposed framework" box of Fig. 1. Our system includes

[2] `https://openwhisk.apache.org/`
[3] `https://knative.dev/`

the following components: we call *computers* the servers with computational capabilities to offer; *clients* the User Terminals (UTs) issuing lambda function requests for the purpose of (partially) offloading their applications; *dispatchers* the entry points of clients to the edge computing domain. In particular, the clients issue their lambda requests to the dispatchers, which forward them to the most suitable computers at a given time based on their local knowledge, then transfer back the response to the clients. Such an exchange is short-lived and it is terminated immediately after completion. The dispatchers do not communicate among themselves or with a centralized entity for scalability reasons, thus the proposed architecture is fully distributed. Compared to the existing alternatives, reviewed in Sec. 2.1, it achieves (i) *low latency*, because there is no centralized entity where all requests are directed; (ii) *scalability*, because every dispatcher only needs to keep the state associated with the in-flight lambda executions of its associated users; (iii) *reliability*, because a single dispatcher failure only affects its associated users. Among the many research challenges associated to the proposed architecture, in this paper we focus specifically on the *online dispatch algorithm* to select the computer that will run a given function at a given time, which is tackled in Sec. 3.2. In the literature of edge computing, this problem involves tasks lasting much longer than in our case, typically requiring the set-up of a Virtual Machine (VM) on a hypervisor running in the edge node, which does not fit the serverless paradigm that we are tackling here. On the other hand, as discussed in further details in Sec. 2.2, some research works have dealt with pure serverless systems, but only when using multi-core servers in a data center, which is clearly a much more controlled and centralized environment than ours.

With regard to the **performance evaluation** of edge computing, in Sec. 2.3 we review the existing approaches, which we have classified into four categories: mathematical models; cloud models; packet-level simulations; testbed experiments. We show that all these approaches are very well suited to capture only a portion of the complexity incurred by an edge computing system, where the nodes may have heterogeneous capabilities and connectivity, and the Key Performance Indicators (KPIs) are affected by both computation and communication aspects, which are hard to capture at the same time. In Sec. 4 we propose a novel framework for the evaluation of algorithms and protocols in an edge computing environment, which surpasses the limitations of the existing alternatives. We argue that this framework, which relies on network emulation, application-based virtualization, and (partial) simulation of computation elements, is general enough to support a wide range of environments. To validate the framework and, at the same time, show the effectiveness of our proposed online dispatching algorithm, we implemented the key components of serverless edge computing and performed an extensive campaign of experiments, whose results are reported in Sec. 5. Our proposed algorithm has been compared with a centralized approach, as commonly found in serverless computing in data centers, and a known online algorithm from the literature [6].

2. State of the art

In this section we review the state of the art in the literature, separately for the main contributions of this work: architecture for the convergence of the edge computing and serverless computing paradigms (Sec. 2.1); distributed dispatching of lambda functions to computers (Sec. 2.2); modeling and large-scale simulation of edge computing environments (Sec. 2.3).

2.1. Converged serverless edge computing architecture

The topic of edge computing is not new in the scientific literature. One of the main trends is to make the orchestration more efficient than that used in cloud systems, which relies on VMs. For instance, the authors in Picasso [7] propose the concept of providing the applications with an Application Programming Interface (API) for the execution of functions in the network, which we have reused here, whereas [8] put forward the use of containers to achieve easier migration than with VMs. On the other hand, in [9], Information-Centric Networking (ICN) is used to realize *Named Function as a Service (NFaaS)*: function execution images, employing an extreme form of containerization called *unikernels* [10], are automatically loaded on ICN-enabled servers based on context and utilization. The main contributions of all these studies are solutions to tune the deployment of application images in the edge nodes, which is complementary to our goal, i.e., to dispatch tasks efficiently at very short scales, by considering a fixed deployment of functions.

Some researchers have also focused on solutions for specific use cases, chiefly the Internet of Things (IoT). A cloud-edge architecture is proposed in [11], which employs micro-services to run computation on heterogeneous edge or cloud nodes: different algorithms are proposed to schedule tasks from the clients, under the assumption that a centralized scheduling engine can estimate at fine grain both the task processing time and the time it will be put into service, which is advocated to be realistic in some conditions, i.e., with *a priori* knowledge on the application algorithms and continuous monitoring of the computation engines. When applying the constraints of our use case to the scheduling policies considered there, they basically collapse into scheduling every task to the computer with shortest expected processing time, which is exactly what we propose in Sec. 3.2. In the same context, in [12] the authors propose to distribute tasks to servers based on Software Defined Networking (SDN) data flow manipulation: this is possible because they assume that all client requests are directed towards the SDN controller, which can thus predict arrivals and assign clients to servers accordingly. Unfortunately, in our approach this is not viable, since we are aiming at a fully distributed system.

2.2. Distributed dispatching of lambda functions

Another branch of research is dealing with a more abstract understanding of the problem we tackle in this work: multi-server scheduling, where tasks issue jobs (with unknown arrival and execution times) that have to be dispatched to a pool of servers, with the objective of obtaining minimum latency. In this context,

it is known [13] that no online algorithm can have a bounded competitive ratio. An *online* algorithm takes decisions per job and cannot keep a job on hold without assigning it to a server, whereas an *offline* algorithm has full knowledge of arrivals of tasks (also including the future ones) and may decide to keep a server idle for some time to maximize the global objective function.

There are at least three important differences between edge computing and the multi-server problem setting. Firstly, the latter assumes that the policy for the execution of jobs in each server can be fully controlled, because the typical target systems are multi-core servers, but we cannot reasonably assume to control scheduling of all the edge nodes, which are highly heterogeneous and shared. Secondly, in an edge system there is no single entry point for the jobs. Lastly, the latencies between edge nodes may vary greatly, because of the heterogeneous communication technologies and since the network devices are also used by the clients for Internet access.

With regard to edge computing in particular, in [6] the authors design an algorithm that minimizes the sum of the latency of the tasks that is $\mathcal{O}(1/\epsilon)$-competitive in the $(1+\epsilon)$-augmented problem. They assume that the scheduler knows the communication latencies and the processing time of every task if executed on a given server. In general this information is not available, but we have implemented in our testbed the algorithm with emulated computers and compared the results to those obtained with our solution in Sec. 5.4.

The authors in [14] also tackle specifically edge computing by defining centralized and distributed algorithms to solve an optimization problem: jointly optimize the processing time and energy consumption at mobile users, where the network is modeled as a network of queues. In the distributed version, the authors study the trade-off between the communication overhead for synchronization and the performance of the algorithm. In this work, since we focus on pervasive applications and the edge nodes are assumed to have limited capabilities, we consider only the limit case where there is *no synchronization at all* between the edge nodes, and show through emulation experiments that this approach is not detrimental compared to the case where there is a single entity making network-wide choices.

Some other works explored the concept of "tasklets", which are similar to micro-services, because they are invoked by clients as functions, and are intended to be used to realize in-network computation with user devices. More specifically, Edinger *et al.* [15] envision a system where computation consumers contact brokers that dispatch them to the most suitable computation producers, who are then contacted directly for the execution of tasklets. In the work above the most important scientific result is a scheduling algorithm that reduces the number of execution failures by estimating the reliability of producers. The problem of task partitioning is studied in [16], along with solutions to reactively/proactively migrate the VMs to minimize the overall task execution time despite failure of devices offering computation. Such contributions are not directly applicable to our case because the computers are devices specifically committed to computation offloading and, thus, can safely be assumed to disconnect or fail very sporadically. Furthermore, the tasklet scenario is flat, and

brokers are introduced merely for scalability reasons, whereas an edge computing network is well structured, with clients logically separated from computers by access network gateways, which we use as dispatchers: in this structure the latter can easily monitor execution of lambda requests, which always pass through them, unlike brokers for tasklets.

2.3. Edge computing modeling and simulation

The performance evaluation of serverless in edge computing is a scientific challenge *per se*. We can distinguish four main categories in the literature.

We consider the pure mathematical approaches first, where a vast number of them is aimed at determining the best allocation of VMs/containers under various constraints. A fairly comprehensive and recent model suited to placing Docker containers on edge nodes is [17], which uses an Integer Linear Programming (ILP) problem formulation. Modeling the edge network as a network of queues is also found in some works, e.g. [14]. While the use of a simplified model makes the problem tractable in mathematical terms, this inevitably hides some aspects, which in the case of devices with limited computation and communication capabilities are crucial to capturing key aspects of the system dynamics, such as the overhead of resource sharing, communication protocol latencies, and processing delays.

Some of these aspects are modeled accurately in the second approach, i.e., cloud-oriented simulator, reviewed more extensively in [18]. The most widely used such simulator is CloudSim [19], which has been extended to also support specifically, e.g., containers [20] and edge computing scenarios [21]. These simulators are aimed at evaluating the relative performance of allocation algorithms, both off-line and dynamic, also introducing a basic simulation of the effects due to networking, especially in the edge-oriented flavors. A key advantage over the pure mathematical approaches is that they can model a wide variety of workloads, also taken from real-life datasets, and still support the performance evaluation of large-scale scenarios at very reasonable computational cost (see, e.g., [22] for a highly-scalable simulation architecture written in Scala). However, to the best of our knowledge, it is not possible with any of them to assess the performance of real applications or system components, because the simulation fully happens within its own realm, with no exchange points with the real world. For the sake of completeness, we mention that the evaluation of real applications using a CloudSim derivative has been attempted [23], but with severe limitations and achieving a level of maturity inferior to the main tool.

This limitation is overcome by using discrete-event packet-level simulators, which most often allow real packets to be injected into the engine, and at the same time offer very detailed model of the physical and networking environment. Examples include Omnet++ [24] and ns-3 [25]. Unfortunately, real-time emulation requires that the simulation engine is fast enough to process all the events, which limits significantly the size of the scenario that can be evaluated.

Finally, the fourth approach is the execution of experiments in a real edge/cloud environment. This direction has been followed to compare existing serverless

frameworks (open source in [26], commercial in [27]). While this provides obviously an environment as close as possible to a production system, there are several disadvantages: experiments tend to be extremely expensive in terms of both time and equipment; it is difficult to reproduce experiments because of the many factors affecting performance in real life, especially if the platforms are not fully owned by the experimenter; the scale of the experiments is limited by the physical resources owned or accessible.

Table 1: Qualitative comparison of different approaches to performance evaluation of edge computing systems.

Aspect	**Math model**	**Cloud simul**	**Packet-level simulation**	**Testbed**	**Proposed framework**
Computation accuracy	-	+	–	++	+/++
Network accuracy	-	+	++	++	+/++
Scenario size	++	+	+	-	+
Real world components	–	-	+	++	+
Reproducibility	++	++	+/++	-	+/++

To overcome the limitations introduced by each respective approach, while relinquishing only a fraction of their advantages, we propose to adopt the following methodology for the evaluation of architectures, protocols and algorithms in edge computing systems, described in Sec. 4. On the one hand, we use mininet[4] for network emulation, which we have customized to suit edge computing environments. Inside mininet, we run actual applications: the overhead of having many instances on a single Operating System (OS) is limited since mininet uses process-based virtualization, based on Linux's *namespaces*. The same has been done already in [28], where the authors describe their wrapper for an easier configuration of edge/fog computing topologies, called EmuFog. On the other hand, we provide a simulator of the computation elements, which allows large-scale environments to be investigated under limited availability of hardware resources to execute experiments. The experiment runs in real time, hence it is possible to mix real and simulated computation elements seamlessly. The proposed framework has been used to validate the performance of the distributed dispatching of serverless functions in Sec. 5.

In Table 1 we summarize the strengths and weaknesses of the performance evaluation approaches found in the literature, plus our framework, in terms of the following key aspects: (i) accuracy in modeling computation processes; (ii) accuracy in modeling network dynamics; (iii) scalability to large-scale scenarios; (iv) possibility to integrate real applications and components; (v) capa-

[4]http://mininet.org/

bility to obtain consistent and repeatable results. As can be seen, our proposal is an excellent trade-off between model accuracy and scalability/usability.

This paper is an extended version of [29], including the following new major contributions: measurement of the average computational cost of the dispatching algorithm (Sec. 3.2.3), complete description of the performance evaluation framework (Sec. 2.3, Sec. 4); two new batches of simulation experiments (Sec. 5.1 and Sec. 5.3).

3. Serverless edge computing

In this section we first describe our proposed framework for the execution of stateless tasks in an edge system (Sec. 3.1). We then design the algorithm for distributed dispatching of such lambda functions to edge servers (Sec. 3.2).

3.1. Architecture

The proposed architecture is illustrated in Fig. 2. As introduced in Sec. 1 it consists of: devices with computational capabilities, called *computers*, which are co-located with networking devices of the edge system. The computers, in general, are not required to have same or similar hardware. Rather, in many cases of practical interest they will have specialized hardware for the execution of some lambda functions, e.g., Graphics Processing Unit (GPU) for Augmented Reality (AR) and video transcoding [30]. All the points of attachment of clients to the network, e.g., Long Term Evolution (LTE) e-NBs or Wireless Local Area Network (WLAN) access points or a mix of them, act as *dispatchers*.

We identify two types of system functions: offline and online. *Offline functions* happen on a long time scale (>= minutes) and are independent of the current lambda transactions being requested by clients. *Online functions* are associated to lambda transactions, thus their dynamics are much faster (<= seconds). We assume that all functions that are not strictly required per lambda execution are pushed into a logically centralized entity, called *controller*. In addition to performing Authentication and Authorization (AA), the controller learns about the existence of capabilities of new computers and dispatchers, respectively, and runs a periodic optimization to modify the lambda functions offered by the computers. This operation is usually called "service placement", and it has received noticeable interest in the research community (see, e.g. [31]), though a definitive solution has yet to be found, especially in the presence heterogeneous hardware. In this work we do not tackle service placements issues and we assume that in between consecutive optimization cycles, the set of images in every computer is constant: lambdas can *immediately* be put into execution upon arrival from dispatchers, with no set-up costs or cold-start effects.

A sequence diagram of a lambda transaction is illustrated in Fig. 3. As can be seen, upon receiving the request for the execution of a given lambda, the dispatcher i selects the most suitable computer k and forwards it the input received; once a response (*res* in the figure) is received from the computer, it is returned to the client. Note that the sequence does not involve the controller,

Table 2: Notation used. The upper part is relevant to the modeling and estimation, whereas the lower part contains the notation for the proposed implementation. The time t is often omitted in the text for readability.

Symbol	Meaning
$\mathcal{L}$	Set of tasks (lambda functions)
$\mathcal{C}$	Set of computers
$\delta_{jk}(t)$	Total delay of task j if dispatched to computer k at time t
$\tau_{jk}(t)$	Networking delay component of $\delta_{jk}(t)$
$p_{jk}(t)$	Processing delay component of $\delta_{jk}(t)$
$\hat{\delta}_{jk}(t), \hat{\tau}_{jk}(t), \hat{p}_{jk}(t)$	Estimated values of the delay components
$S_j(t)$	Size of the request of task j at time t
W_τ / W_p	Size of the moving window to estimate the network / processing delay
W_S	Set of quantized request sizes considered
$u_{jk}(t)$	Load reported by the computer k executing task j at time t
$\alpha_k^\tau(t), \beta_k^\tau(t)$	Intercept/slope to estimate the networking delay to reach computer k at time t
$\alpha_{S'jk}^p(t), \beta_{S'jk}^p(t)$	Intercept/slope to estimate the processing delay of task j to be dispatched to computer k when receiving a request quantized as size S' at time t
T_τ / T_p	Timeout values for the collection of network / processing delay samples

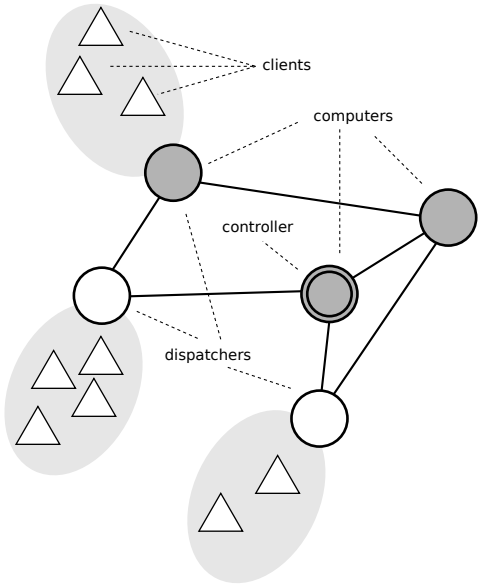

Figure 2: System architecture.

which only operates with offline functions. In the next section we discuss the matter of selecting the best computer for the execution of a lambda function, called the *distributed lambda dispatching problem*. The notation used throughout the paper is summarized in Table 2.

3.2. Distributed lambda dispatching

In the following we describe the proposed algorithm to solve the distributed lambda dispatching problem, i.e., selecting the best computer $k \in \mathcal{C}$ to dispatch an incoming lambda function request $j \in \mathcal{L}$ ($\|\mathcal{L}\| = L$) at time t. As explained in the section above, every dispatcher knows which computers in $\mathcal{C}$ offer microservices of matching type from the controller, and this set changes very slowly compared to the dynamics of lambda dispatching. Let $\delta_{jk}(t)$ be the delay of job j if dispatched to computer k at time t. As illustrated in Fig. 3, the total delay is the sum of the networking delay $\tau_{jk}(t)$ and the processing time $p_{jk}(t)$. The former, i.e. $\tau_{jk}(t)$, includes the transmission time and queuing delays in all intermediate transmission hops. The latter, i.e. $p_{jk}(t)$, is the time required for processing the lambda function on the computer, which depends on both its hardware capabilities and the other concurrent tasks sharing the resources with j until its completion.

A good policy to solve the dispatching problem is Shortest Remaining Processing Time (SRPT), which is known in the literature and is widely employed in multi-server schedulers because it has a bounded competitive ratio, it is simple, and it is resilient to estimation errors of the processing time [32]. Ideally,

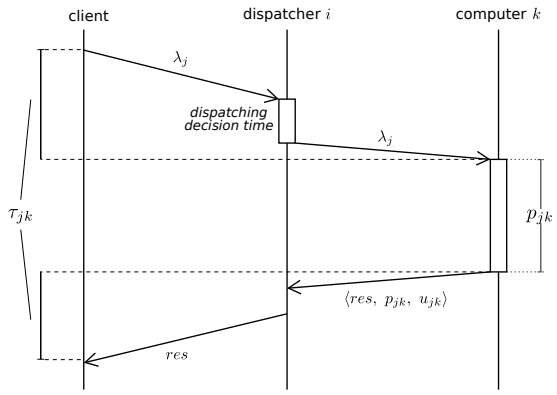

Figure 3: Lambda request/response sequence.

with SRPT the dispatcher selects computer $\bar{k}$:

$$\bar{k} = \arg\min_k \{\delta_{jk}\} = \arg\min_k \{\tau_{jk} + p_{jk}\}, \tag{1}$$

where we have dropped the time t reference to simplify notation. Of course, in practice we do not know in advance the network and processing delays, therefore we define $\hat{\delta}_{jk}$ as the *estimated* delay of job j if dispatched to computer k, and similarly for $\hat{\tau}_{jk}$ and $\hat{p}_{jk}$. To emulate as closely as possible the behavior of an ideal SRPT scheduler, we need a solution to estimate $\hat{\tau}_{jk}$ and $\hat{p}_{jk}$, which has to be: i) *simple*, because the computational capabilities of the dispatchers may be limited, ii) *fast*, because the decision time reflects on the application delay, and iii) *subject to uncertainty*, because the dispatcher uses only local information. In the following sections we tackle the estimation separately for the networking delay (Sec. 3.2.1) and the processing delay (Sec. 3.2.2), before composing the overall dispatch algorithm (Sec. 3.2.3).

3.2.1. Networking delay estimation

To estimate the networking delay τ_{jk} we propose that computers piggyback the processing time into the responses to the dispatchers. This way, the dispatcher can sample the networking delay with every computer by simply keeping track of the overall time required for the job execution. It is not even required that the dispatchers and computers are time synchronized, since the time difference is relative and computed individually by every dispatcher. A simple, yet effective, linear regression model is sufficient to estimate and predict precisely the networking delay, since the latter consists of a fixed component, depending only on the network topology and communication technologies used, and a variable component proportional to the amount of data transmitted.

Any more sophisticated approach, which fits better the specific deployment, can be used without affecting the overall architecture proposed.

3.2.2. Processing time estimation

The estimation of the processing time p_{jk} is more challenging. In general, predicting the processing time of a non-trivial algorithm executing on a shared general-purpose computer is extremely difficult, because the result depends on a huge number of factors and contingent conditions. It is beyond the scope of this paper to investigate the issue in full details, as done for instance in [33], where the authors propose a Machine Learning (ML)-based cloud task execution prediction framework. Furthermore, accurate prediction requires application- and scenario-specific details: for instance in [34] tasklets (see Sec. 2.2) are matched to computation resources based on a learning-based approach, which however requires the source code of the applications to be passed through a static profiler for feature extraction. On the other hand, we propose the following practical scheme that can be used in general cases, i.e., without *a priori* knowledge of the algorithms (workflow, code, etc.) and the internal details of computers (OS, scheduling policy, etc.).

Firstly, we assume that every computer k piggybacks on the responses the current system load, indicated as u_{jk} for lambda function j. This assumption is rather weak since every modern OS is able to provide effortlessly such information to its applications[5].

Secondly, we observe that in practice it is reasonable to expect for any given lambda function j and computer k, an increasing relation between the processing time p_{jk}, with given input size S_j, and the load u_{jk} in the recent past. In other words, if a computer has been heavily loaded in the last few seconds, then it is likely that it will be still so in the near future, thus extending the execution time of any new job assigned. Also, for a given computer, the processing time will generally increase as the input size increases, all other conditions (e.g., the load) being the same. Therefore, we propose that every dispatcher keeps track of the past processing times occurred, together with the lambda input size and the load reported by the computer. This provides all dispatchers with the following 2D mapping for any given lambda and computer, which can be used to extrapolate $\hat{p}_{jk}\colon \langle S_j, u_{jk} \rangle \to p_{jk}$.

Lastly, we assume that the processing time p_{jk} increases linearly with the lambda request size S_j. Unlike the two assumptions above, this one is restrictive and we cannot expect it to be true for every possible application and use case. However, we keep it here as a working assumption, because it does not preclude any other more sophisticated approach to be used, while keeping the rest of the architecture and algorithms unchanged. In practice, for a target deployment, we foresee the following approach. If available, *a priori* knowledge should be exploited to determine a suitable processing time model, so as to use a matching estimation algorithm since the beginning. Otherwise, a linear model, as assumed in the rest of paper, is a reasonable starting point for many applications of

[5]Though it may require careful consideration if VM or containers are involved since the "system" load may not correspond to the achievable load because of virtualization/isolation mechanisms.

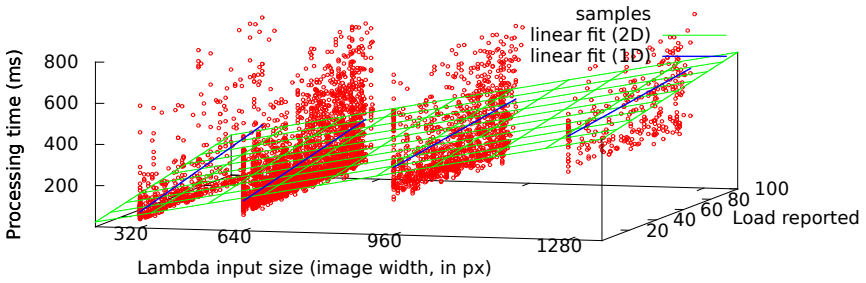

Figure 4: Example of 2D and 1D linear fitting of processing data values in the dispatcher, for each computer k and lambda function j.

practical interest: the model can be then improved based on the analysis of live data.

Therefore, let us assume that p_{jk} is a linear function of both the lambda input size and the computer load. In this case, estimation of the processing time can be done by finding the plane that best fits the population of samples collected. Since this fitting must be updated at every new lambda, we further propose to reduce the computational complexity by quantizing the lambda input size into a set of discrete values, then finding a 1D linear fit as a function of the load values only. This process is visualized in Fig. 4, which shows the population of processing time values as a function of the lambda input size and load reported collected by a dispatcher during one of the experiments described in Sec. 5.2 below. The details of the experiment are irrelevant at this stage, but we anticipate that real computational offloading, i.e., face detection on static images, is being performed. As can be seen, there are four possible image sizes, yielding an equal number of lambda input sizes, and the processing time increases as either the input size or the load reported increases. In the 3D plot we show both a linear regression of the 2D plane and four 1D linear regressions, one per lambda input size: the 1D linear regressions are very close to the 2D plane, but they can be achieved at a fraction of the time complexity, which suggests that our proposed approach is valid.

3.2.3. Overall dispatching algorithm

To summarize, every time a lambda function j with a request of size S_j is correctly executed by computer k, which reports load u_{jk}, yielding an *a posteriori* processing time p_{jk}, the dispatcher performs the following housekeeping operations:

1. measure the communication latency τ_{jk} as the difference between the overall lambda execution time, which is a local information, and p_{jk};
2. add $\{S_j \rightarrow \tau_{jk}\}$ to a moving window of W_τ samples and find the intercept/slope values $\alpha_k^\tau / \beta_k^\tau$ that best fit them;

3. quantize S_j as S' to the closest value among the W_S possible ones and add $\{u_{jk} \rightarrow p_{jk}\}$ to a moving window of W_p samples and find the intercept/slope values $\alpha^p_{S'jk}/\beta^p_{S'jk}$ that best fit them.

By design, measurements are always collected as the result of the dispatching algorithm selecting a computer as the destination of a lambda function execution request. This sort of *passive polling* guarantees high scalability as the number of computers and dispatchers grows, but it has a side effect possibly leading to sub-optimal selection: once a computer becomes affected by a bad reputation due to possibly temporary overload conditions, either because of concurrent tasks or network traffic, such a status may never be cleared in the future because the dispatcher is unlikely to select it as the best choice. To prevent such a form of starvation, we associate a lifetime to the measurements: after a timeout T_τ (T_u) the values collected regarding the communication latency (processing time) estimation process are discarded, and the computer's state is restored afresh as if it had just entered the edge computing domain. In a production environment where an optimization process adapts the computational/networking resources over time, the lifetime-based approach could also be substituted by an event triggered on the dispatchers by the orchestration layer.

The overall dispatching algorithm for an incoming lambda function of type j, whose input is S_j, quantized as S', consists of finding the destination computer $\bar{k}$ s.t.:

$$\bar{k} = \arg\min_k \left\{ (\alpha^\tau_k + \beta^\tau_k S_j) + \left(\alpha^p_{S'jk} + \beta^p_{S'jk} u_{jk} \right) \right\} \tag{2}$$

To achieve high scalability with non-specialized hardware, it is important that the dispatching algorithm remains as simple and fast as possible as the number of computers and lambda functions grow. The *worst-case computational complexity*, in both space and time, of the main algorithm components is reported in Table 3, where the *house-keeping* rows refer to operations that are carried out upon receiving a successful response from a computer. We briefly recall the notation used in the table: L is the number of possible lambda functions, C is the number of computers in the edge network, and W_τ, W_S, and W_p are the internal parameters representing the number of communication latency samples kept per computer, the number of quantized input sizes, and the number of processing times kept per lambda per computer, respectively.

Table 3: Worst-case computational complexity analysis.

Algorithm	**Space**	**Time**
Communication latency house-keeping	$\mathcal{O}(W_\tau C)$	$\mathcal{O}(W_\tau)$
Processing time house-keeping	$\mathcal{O}(LW_pW_SC)$	$\mathcal{O}(W_p)$
Dispatching	–	$\mathcal{O}(C)$

On the other hand, to quantify the *average computational complexity* of the algorithm we have carried out the following testbed experiments. We have

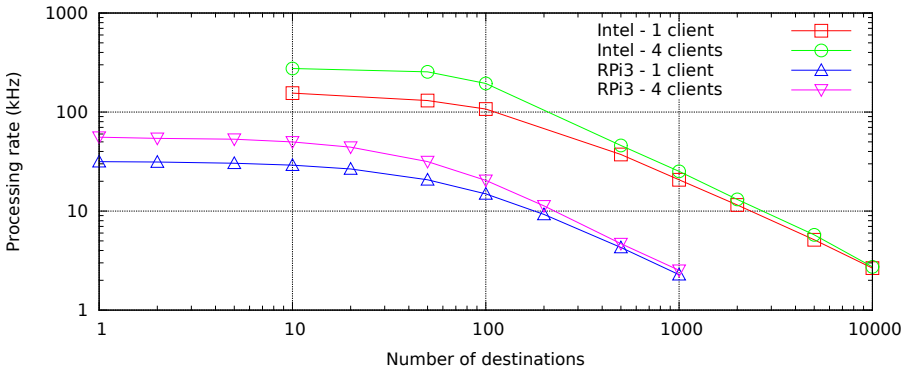

Figure 5: Dispatcher processing rates per core, with a RPi3 and an Intel server.

run a single instance of a dispatcher, whose internal structure and implementation details are illustrated in Sec. 4.4, on a Raspberry Pi 3 Model B (RPi3), which is "representative of a broad family of smart devices and appliances" [35], and on an Intel application server with Xeon CPU E5–2640 v4 at 2.40GHz. Even though the application can exploit parallelism on Symmetric Multiprocessing (SMP) architectures by spawning multiple threads, for the purpose of the experiment we have restricted the execution on a single core[6]. In the experiment we have installed a given number of possible destinations of a special lambda, for which the dispatcher replies immediately to the client without actually reaching a computer. However, all other phases of the dispatching algorithm, including the computation of Eq. (2) and the update of all the internal data structures and timers, are performed exactly as with regular lambda functions. We have run experiments with 1 and 4 threads spawned in the dispatcher, and executed a matching number of clients repeatedly asking for the special lambda function to be executed.

In Fig. 5 we report the processing rate achieved in the different combinations, which gives an upper bound of the performance of the dispatcher (per single core) and a quantitative measure of the computational cost of the proposed algorithm as the number of destinations (C) increases. For each combination of parameters we have executed 10 repetitions, but we do not report error bars in the plot because the variance was negligible. As can be seen, in all cases the processing rate decreases linearly in the log-log plot, which means that the average computational complexity increase w.r.t C is sub-linear, unlike the worst-case computational complexity, which is in fact often considered too pessimistic. This provides for a smooth scalability of the dispatcher as the edge computing size, i.e., the number of computers, increases. In absolute terms, the Intel server-grade Central Processing Unit (CPU) obviously performs much bet-

[6]This can be done in Linux using the so-called *affinity* mechanism.

ter than RPi3's ARM CPU, and achieves a processing rate significantly greater than 1000 Hz, which means less than 1 ms overhead per function call, with up to 10,000 possible destinations. However, also the RPi3 incurs a reasonable small overhead in a range that is certainly relevant for many practical deployments.

In the supplementary material we also include the CPU utilization measured during the experiments.

4. Simulation framework

As illustrated extensively in Sec. 2.3, to the best of our knowledge, the existing solutions to evaluate the edge computing performance have some limitations when realistic modeling of both *connectivity* and *computation* is required. In this section we propose a novel framework that overcome these limitations. The framework has been developed originally for the performance analysis of the serverless edge computing architecture described in Sec. 3, but it can be used "as is" for the evaluation of generic algorithms and protocols in a wide range of scenarios.

Our solution builds on network emulation and lightweight virtualization, combined with simulation of the computation processes. Therefore, all system components are instanced as Linux applications running inside `lxc` containers, which is a form of process-based virtualization, incurring a negligible overhead compared to running the same application on the host OS. This allows the experimenter to take into account in a realistic manner all the effects due to the communication protocols in use, as well as including in the analysis several phenomena that are very difficult to capture using simulators, e.g., inter-process communication overhead and caching. Furthermore, network emulation can be realized efficiently using mininet, which is known to scale well to large topologies with thousands of nodes on a single server[7].

The use of real components in experiments, however, does not mean that necessarily *all* applications must be real ones. In fact, we argue that in many cases the use of real edge applications is not needed. This is the case of our distributed dispatching system for serverless edge computing, described in Sec. 3, which focuses on the forwarding of the functions, but it is agnostic of the actual applications being run by the clients. Since edge applications are very likely to be a choke point for the execution of large scale experiments, because offloading is especially appealing for computationally-intensive tasks that cannot be executed efficiently on devices, by simulating computation we can retain all the advantages of a real testbed with a fraction of the hardware required. This, in turn, opens the door to easy automatization of the execution of the experiments, simplified collection/analysis of results, and straightforward repetition of experiments.

[7]If this is not sufficient, its extension MaxiNet (`https://maxinet.github.io/`) allows the distribution of mininet "islands" over multiple servers, thus enabling theoretically to scale up experiments to any arbitrary size, provided that sufficient computational and network capabilities are available.

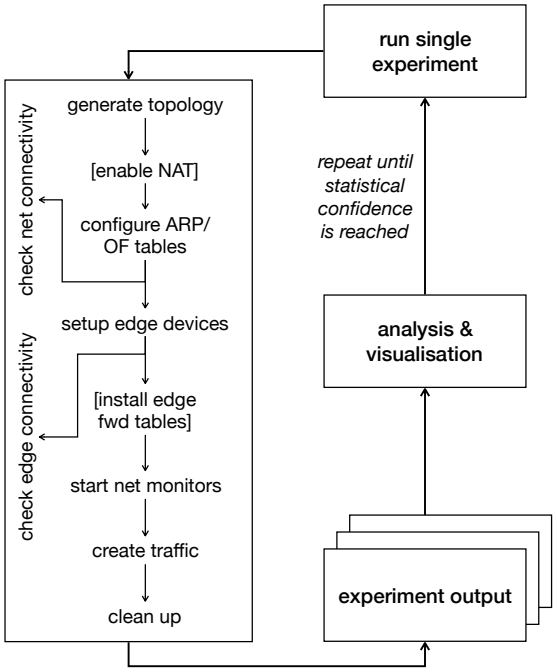

Figure 6: Workflow of a generic experiment.

In the remainder of this section we first describe the experiment workflow (Sec. 4.1) and the computation element simulation model (Sec. 4.2), which are both general and application-agnostic. Afterwards we focus on the serverless-specific components of our framework: we illustrate the image manipulation computer (Sec. 4.3), used in Sec. 5 directly, i.e., instances are executed as part of the experiments, and indirectly, i.e., to calibrate the configuration parameters of the computation element simulation model, and finally the serverless edge computing dispatcher (Sec. 4.4).

4.1. Experiment workflow

In Fig. 6 we illustrate the generic experiment workflow: the execution of a single experiment is done by launching a Python script, which includes a number of generic modules developed as part of our framework, interacting with mininet via a Python API. The modules implement the logic to realize all the steps reported in the left hand side box in Fig. 6, including the topology generation according to models suitable for edge computing environments and the configuration of the OpenFlow (OF) and Address Resolution Protocol (ARP) tables in switches to ensure network-layer connectivity. Every intermediate step may be customized by the experimenter by overriding the relevant methods of the main experiment class to suit their specific needs. Common-use metrics, such as

per-switch throughput, are collected by default and made available at the end of the experiment, together with custom metrics. The analysis and visualization of the experiment output is instead beyond the scope of the framework, since it ultimately depends on the actual environment.

Our framework is general enough that it can be used to simulate a multitude of different edge computing environments, also including interactions with real world applications living outside the virtualized mininet environment through an appropriate Network Address Translation (NAT) configuration. We have used the framework to validate and assess the performance of our distributed architecture for serverless edge computing with real and simulated computation elements, in a vast range of network topologies, using different methodologies (transient analysis, steady-state with replications, Monte Carlo).

4.2. Computer simulation model

We now present the model of the *simulated computer*, which is an application that performs arbitrary functions without really implementing any algorithm, but rather simulates internally the execution of the currently scheduled lambda functions to mimic the behavior of an SMP multi-container edge server. Tasks are served according to a First Come First Serve (FCFS) non-preemptive policy. Every task is associated to requirements in terms of both computation (number of operations) and memory (bytes). In the simulated computer implemented we have used a linear model where the number of operations (or memory) required is the sum of a constant offset and a value that is proportional to the input of the lambda function request, in bytes. In the supplementary material we show that this model is able to capture very well the response time of the image manipulation computer described in Sec. 4.3. Different offset/slope values can be used for different lambda functions in different computers: this allows to simulate computers that have heterogeneous characteristics, such as equipped with GPUs/Vision Processing Unit (VPU) better suited to graphics/ML jobs than general-purpose CPUs. The number of operations is used to determine the simulated processing time, which also depends on the number of cores and the CPU speed. Multiple tasks can share the available cores, provided that the container on which they are running have sufficient *workers* available, otherwise the tasks are put into a waiting list. On the other hand, the memory required is used to block tasks whose requirement would exceed the residual memory available, which is the difference between the total amount installed (in simulation) and the sum of the memory requirements of all active tasks. If a task is blocked because of insufficient memory, no other task is put into execution until it is eventually made active, to avoid starvation. Different models to compute the computation/memory requirements, as well as different scheduling policies, can be easily implemented, should they be needed to better model a given platform or application under evaluation.

To better explain the behavior of the simulated computer we now illustrate two example simulations, both with three incoming tasks. For simplicity we assume there is a single simulated core. All the lambda functions are served by the same container, i.e., they are of the same type, with two workers. In the

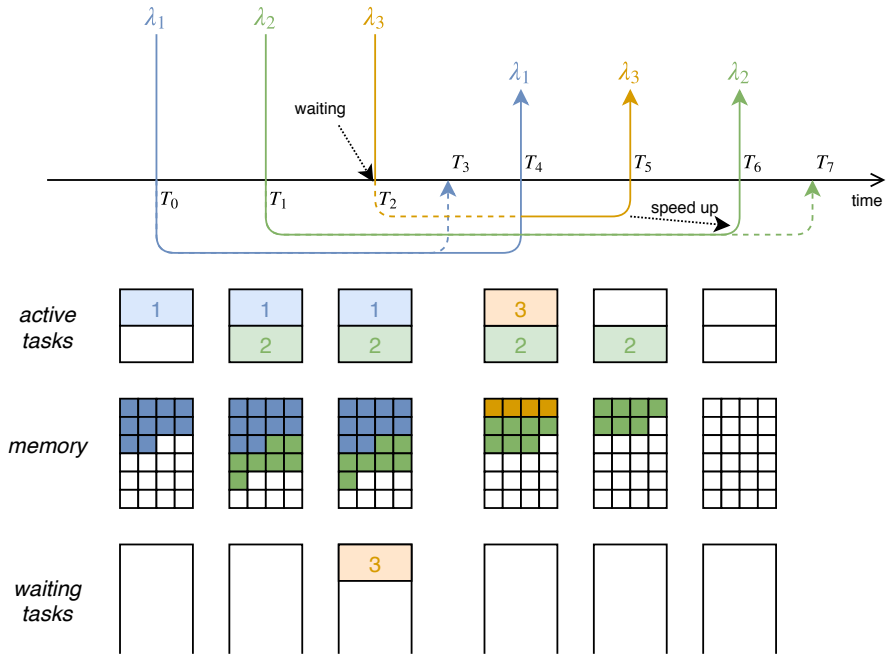

Figure 7: Example of processing simulation.

first example, in Fig. 7, memory is not a limitation and it is not considered (it is the subject of the second example). At T_0 the first task λ_1 enters the system; based on its input size and the offset/slope configured for this container in this computer, the completion time T_3 is assigned: if no other tasks arrived, that would be the time when λ_1 returns. However, at T_1 a second task λ_2 arrives: in addition to assigning a completion time to it (T_7), the simulated computer also modifies the completion time of λ_1 to T_4 because from now on it has to share CPU cycles with λ_2. When a third task λ_3 arrives, it is put into the waiting list because there are no workers available; this situation changes only as λ_1 leaves the system, when λ_3 is put immediately into execution: note that the completion time of λ_2 which is still active, does not change because the number of *active* tasks remains the same. On the other hand, as λ_3 completes its execution, λ_2 becomes the only active task, which makes the computer shorten its completion time from T_7 to T_6.

In the second example, in Fig. 8, the tasks have exactly the same characteristics and arriving times, but we now have a smaller memory. Because of this tighter constraint, as λ_2 arrives at T_1, it cannot be made active because memory is insufficient. Therefore it is put into the waiting list, while λ_1 enjoys full CPU power, hence no completion time advance like in the first example. When λ_3 arrives at T_2, its memory requirements *would* fit into the residual capacity, but

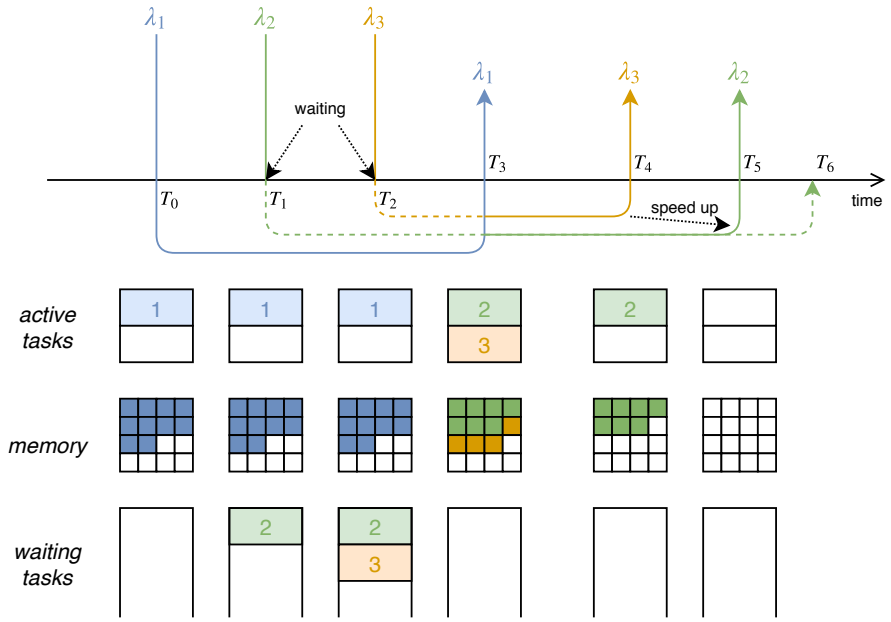

Figure 8: Example of processing simulation, with memory-bound tasks.

it is put into a waiting list together with λ_2 because we enforce a FCFS policy. As already mentioned, this also automatically avoids starvation of tasks with large memory requirements. As soon as λ_1 completes at T_3, both waiting tasks become active, and the simulation then proceeds as in the first example.

Quite clearly, since the simulation must happen in real-time, there are limitations to the processing rates that can be achieved by the simulated computer. We now show in a quantitative manner, with the results from an experiment, that the range of operation of the simulated computer are very reasonable. We run an instance of a simulated computer on a single core of an Intel Xeon CPU E5–2640 v4 at 2.40GHz (same machine used for the dispatcher results in Sec. 3.2.3 above and for the experiments in Sec. 5 below). We are interested into measuring the error introduced by the simulated computer as a function of: the number of cores simulated and the processing rate. To this aim, we set up a number of clients equal to the number of simulated cores, continuously requesting the execution of a lambda function configured with offset/slope such that its processing time is 1 ms, 10 ms, and 100 ms, respectively. Memory limits do not constrain execution of applications. In these conditions, we expect that the response time will be exactly the same as the nominal processing time, because every task has a dedicated simulated container/core for its execution: every deviation is due to the *physical* CPU of the host not being able to keep the pace with the simulated events. In Fig. 9 we plot the *relative execution*

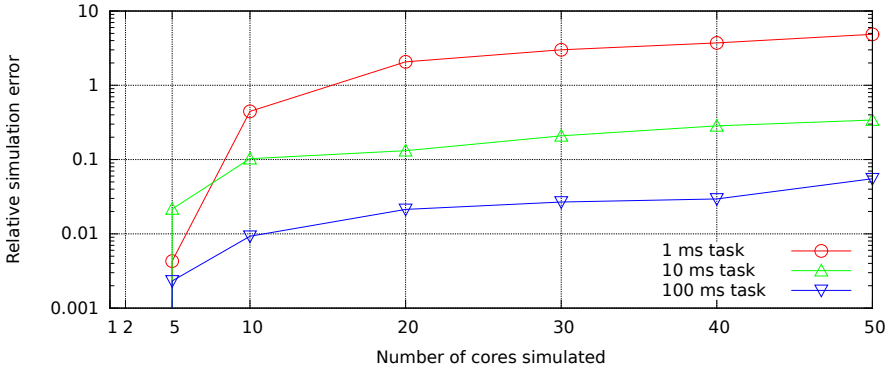

Figure 9: Relative execution error of the computer simulator as the number of simulated cores increases.

error, defined as the ratio between the actual processing time and the expected processing time minus 1. Since the actual processing time is always greater than the expected value, because a CPU may only lag in time, the relative execution error is always positive: for instance a value of 0 means that there is perfect match between simulated and actual processing times, whereas a value of 0.1 means that the computer is introducing an error in the order of 10% of the processing time. In the supplementary material we also report the physical CPU utilization during the whole experiment. As can be seen in the plot, a single physical core is able to simulate with small error a server equipped with many cores, up to 50 in our experiments, provided that the processing time is 10 ms or higher. For instance, this means simulating $50 \times 20 = 1000$ containers on the server we have used, which has 20 physical cores. On the other hand, if simulation with fine-grained granularity of a multi-core machine serving such shortly-lived tasks is required, it will be necessary to allocate sufficient physical CPU cores to the computers to obtain accurate results at high processing rates.

As already introduced, this simulated computer is very important for performance evaluation purposes: (i) it allows to scale experiments up to large networks without requiring prohibitive computational capabilities, (ii) it enables the execution of sensitivity analysis studies, having a totally known and controllable response, and (iii) finally it allows the implementation of the comparison algorithm proposed in [6] because it can predict the completion time of jobs (if no other arrives meanwhile).

4.3. Image manipulation computer

In addition to the simulated computer described above, for the purpose of evaluating our serverless edge computing architecture with a realistic application of practical interest, i.e., mobile AR, we implemented an *image manipulation computer* that actually performs face or eyes detection using the OpenCV li-

brary[8]. This also shows how our performance evaluation framework can work with real applications, simulated ones (as described in Sec. 4.2 below), or any mix of the two. However, we are not interested in the details and challenges associated to the specific detection algorithms, for which we refer the interested reader to, e.g., [36].

The OpenCV library is able to use concurrently multiple CPU cores to reduce the processing time. The maximum concurrency level can be set at run-time, and in the experiments we use this feature to artificially limit the computational capabilities of computers.

With regard to face detection, the edge client on the device sends a picture to the dispatcher selecting `face_detection` as the lambda name. The dispatcher then forwards the request to a suitable computer, which replies with the rectangles enclosing all the faces found, if any. The response time is the time between when the edge client issues the lambda request and when it receives back the response. If detection of eyes is also requested, then the edge client, for every face found in the first step, crops the original image based on the rectangle coordinates and issues another lambda request of type `eyes_detection`, which is also dispatched as before to a suitable computer. In this case the response time is measured by the client between when the initial face detection is requested and when the last eyes detection response is received.

Table 4: Response times of the lambda functions used in the experiments with an image manipulation computer for face detection.

Picture size	**Size (bytes)**	**Comp. only (ms)**	**With network (ms)**
320×240	52830	43 ± 9	60 ± 10
640×480	175332	101 ± 10	218 ± 26
960×720	353230	181 ± 13	446 ± 47
1280×960	560103	301 ± 13	744 ± 45

We report in Table 4 the response times of face detection with the set of pictures used in the performance evaluation, ranging from 320×240 to 1280×960, when using up to two CPU cores. We report the times both when called directly (*Comp. only* column in the table) and when the lambda function is invoked by a client to a computer via a network link (*With network* column in the table) emulating a typical WLAN access network. The variance of results is rather high due to the ML algorithm used in the OpenCV library for detection.

4.4. Dispatcher prototype

We conclude this section by describing the online lambda function dispatcher implemented in our framework, which can be represented using the Finite State Machine (FSM) in Fig. 10. As illustrated, the dispatcher continuously waits for

[8] `https://www.opencv.org/`

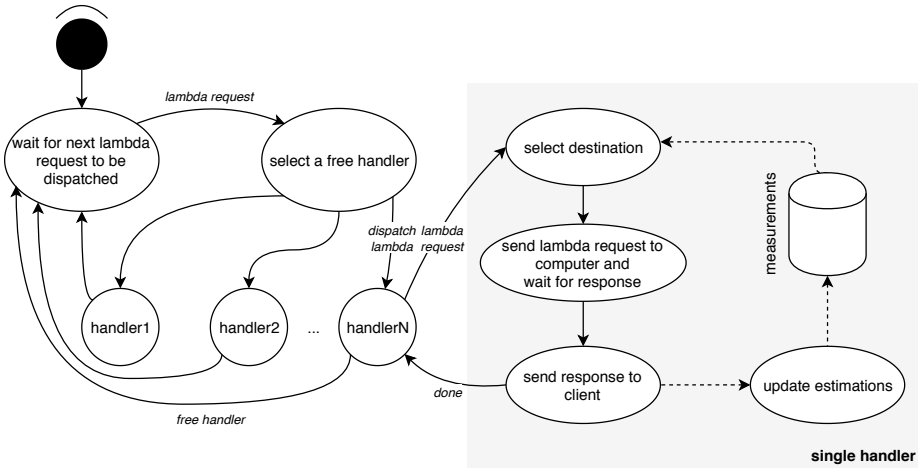

Figure 10: Finite state machine of a dispatcher.

new lambda execution requests from clients. Once a new one arrives, one of the idle handlers takes care of its entire execution until a response is returned to the client before it can be considered idle again. A handler initially selects the destination based on Eq. (2), then forwards the lambda request to the computer selected and waits from a response, which is eventually forwarded to the client. Afterwards, the handler uses a locally measured delay together with the processing time and load indication piggybacked by the computer on the response to perform the so-called house-keeping operations in Sec. 3.2.3. Both the latter and destination selection use data structures that are in common for all the handlers, which may limit in principle the degree of parallelism that can be achieved as the number of handlers is increased, especially for lambda functions with a short execution time.

We implemented the dispatcher in C++ and the execution of lambda functions has been realized by means of REST interface methods using Google's gRPC[9], which is a mature industrial-grade communication protocol using Hyper-Text Transfer Protocol (HTTP), with messages serialized with Google's protobuf library[10], which is lightweight and portable. A *lambda request* contains: the name, that is used to identify the container or run-time environment to be used; the input, that is opaque to the edge computing components; and a flag that, if enabled, tells the dispatcher not to actually forward the lambda but reply immediately with an estimate of the time it would be required to carry out the function. This last field could be used by a client associated to multiple dispatchers to decide where to request execution. Think for instance of a user

[9] `http://grpc.io/`
[10] https://developers.google.com/protocol-buffers/

with a smart phone connected to a mobile network and a WLAN, both offering edge computing services. A *lambda response* contains: a return code specifying what went wrong, if anything; the output; the Uniform Resource Locator (URL) of the computer actually carrying out the computation; the time required for the execution of the lambda; the response, opaque to the edge computing components; a short-term average load of the computer before the execution of the lambda function.

In the prototype, our applications, written in C++, call directly the REST methods on the dispatcher to which they are attached, but integration with any other high-level programming language is possible since the gRPC library is platform and language independent. To simplify the discovery phase, we assumed that every computer and dispatcher registers itself to the controller at a known URL. Finally, to achieve interoperability in a real deployment we envisage adopting standard APIs, such as those defined by the ETSI MEC [37]; such an opportunity is being currently investigated.

5. Performance evaluation

In this section we evaluate the distributed lambda dispatching algorithm with the proposed performance evaluation framework in three setups: an artificial scenario in Sec. 5.1, configured in different flavors to show the flexibility of our simulation framework to model very heterogeneous conditions, as well as to introduce the main characteristics of the proposed distributed dispatching algorithms; a small-scale network of edge nodes, equipped with real face detection capabilities in Sec. 5.2 (this set-up is also used for an assessment of the sensitivity of results to the main internal parameters in Sec. 5.3); a more realistic large-scale environment where we have used simulated computers (Sec. 5.4).

The experiments have been run on a Linux Intel Xeon dual socket workstation, with Intel Hyper Threading disabled, that was not used by any concurrent demanding application. During all the experiments the computational processing demands were never significant enough to suggest that the results might be affected by a hardware limitation of the host used. In all the experiments we have used $W_\tau = W_p = 100$ based on preliminary calibration experiments: we do not report these results because they would give the reader little insight, however we have verified *a posteriori*, using the method of $2^k r!$ analysis [38], that the results are not affected, in a statistical sense, by setting these parameters to 50% and 200% of the value above, see Sec. 5.3 below for full details. The value of W_S depends on the specific experiment and is indicated in the respective sections below. Unless otherwise specified, every experiment has been repeated 10 times: in the plots we report the 95% confidence interval over all the replications.

For all the scenarios below, additional results are available as part of the paper supplementary material.

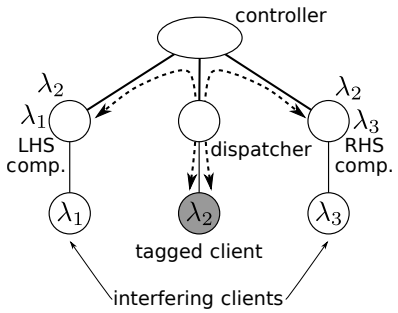

Figure 11: Network topology used in the resource limitations experiments.

5.1. Exploring resource limitations

In this section we show that our proposed framework can adapt to a wide range of environments: as described in Sec. 4 this is very important for the performance evaluation in edge computing environments, since the latter span across several markets / vertical industries, each with very specific and heterogeneous characteristics. We use the simple network topology illustrated in Fig. 11, in which the thick edges have 100 Mb/s bandwidth / 1 μs latency, and they connect a root node (also hosting the controller) to the edge nodes, whereas the thin edges have 25 Mb/s bandwidth / 100 μs latency, connecting the clients to their respective edge nodes. The left-hand-side (LHS) computer offers serverless functions λ_1,λ_2 and the right-hand-side (RHS) computer offers λ_2,λ_3; every client issues lambda function requests of the same type indicated in the figure: as a result of these assumptions, the so-called interfering clients on the left and right of the network may be served only by their respective computers, while the tagged client in the middle has its requests served by the LHS or RHS computer depending on the decisions of its dispatcher.

We have configured the computers to give the same response times, on average, as a real image manipulation computer for face detection, reported in Table 4. The tagged client issues lambda request with size corresponding to a 320×240 picture every 200 ms on average (actual inter-time between consecutive requests is drawn from a uniform r.v.); on the other hand, the interfering clients, when present, use all possible picture sizes in the table with an average inter-time of 1 s. We simulate four cases: (i) Baseline: without interfering clients, only to establish a reference of the performance metrics; (ii) CPU: we make the two computers unbalanced in terms of computation power by reducing to $\frac{1}{3}$ the number of RHS's simulated cores; (iii) Memory: we make the two computers unbalanced in terms of memory by specifying a limited memory for RHS, while the memory of LHS never constrains execution of tasks in this scenario; (iv) Network: we simulate background traffic between the root node and the LHS Computer by transmitting bursts of TCP exchanges lasting for 3 s every 5 s.

In all the four cases we compare the following schemes:

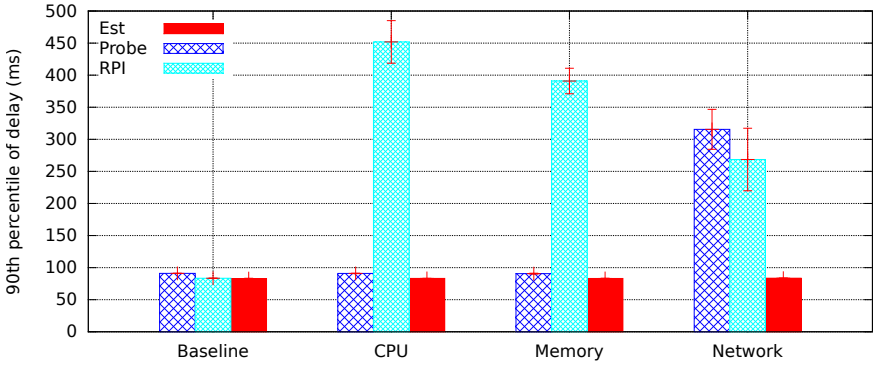

Figure 12: Resource limitations: 90th percentile of delay.

Est : our proposed dispatching algorithm described in Sec. 3.2.3.

Probe (inspired from [6]): a centralized dispatcher co-located with the controller polls each computer to retrieve the execution time required, then selects the one that advertised the smallest value; the implementation of Probe with simulated computers (see Sec. 4.2) is straightforward since they always know precisely the execution time of any incoming task, provided that no other jobs arrive.

RPI (Relative Performance Index, inspired from [39]): the dispatcher uses a Weighted Round Robin (WRR) scheduler to select the destination of incoming lambda function requests, with the weight being equal to its zero-load response time, normalized in range whose bounds are given by the fastest and slowest computer, respectively; the weights are computed in an initial profiling phase during which the dispatcher executes every lambda function on every computer, while no other task is being executed on it.

Clearly, both Probe and RPI cannot be implemented in a real deployment and they are to be considered only as optimistic performance reference: on the one hand, Probe requires the execution time of any task to be known *a priori*, but this information is generally not available; on the other hand, the initial profiling of RPI requires that the edge computing system is unavailable throughout this phase, which may be unfeasible in a live system.

In Fig. 12 we show the 90th percentile of the delay, which is defined as the time between when the client issues a lambda request and when it receives the response from the dispatcher. In the Baseline case the delay is the same for RPI and Est, but slightly greater with Probe, due to the additional delay of polling the computers to check which one would provide the shortest processing time. In the CPU and Memory cases, the 90th percentile of delay increases significantly with RPI because the latter blindly balances between the two computers, regardless of their current state. On the other hand, our proposed dispatching algorithm, using only local estimated information, is as efficient as a centralized

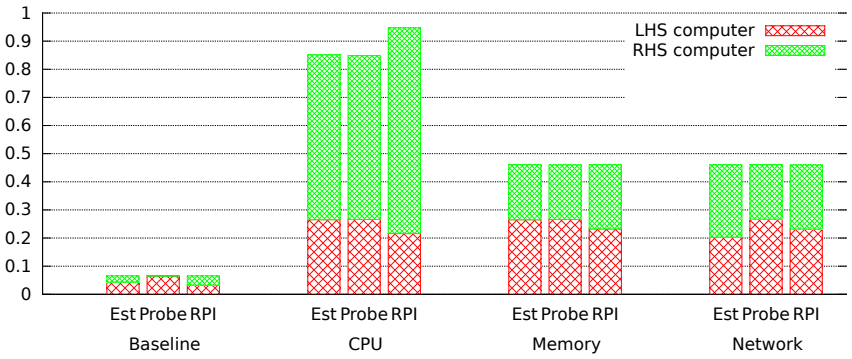

Figure 13: Resource limitation: average load.

version that uses reliable information from the computers themselves. In the Network case, Est outperforms both comparison schemes since it also includes in its algorithm (see Eq. (2)) the communication latency.

In Fig. 13 we provide additional insight on the system dynamics by showing the average load of the computers, defined as the fraction of time that the simulated cores are actively processing tasks. First, we note that with Baseline the Probe scheme uses only one computer: since they are identical and there are no other clients than the tagged one competing for resources, polling deterministically returns the same value on both computers. Second, in the CPU case, the RHS load is higher because it has less simulated computational resources. Finally, in the Memory and Network cases, the LHS computer's load is respectively the same with both Probe and RPI: this is because both solutions take into account only the processing time, either statically identified in a profiling phase or dynamically requested from the computers, which yields worse performance than Est, which instead is more flexible in adapting to the different computational capabilities and serverless/background traffic conditions.

5.2. Small-scale experiments

In this scenario we target a local edge system providing computation offloading services for face detection (see Sec. 4), to mobile nodes via edge nodes in close proximity, as illustrated in Fig. 14. The links between the edge nodes have 100 Mb/s capacity with 1 μs latency, whereas the clients use 25 Mb/s capacity / 100 μs latency links to connect to them. For the purpose of differentiating them, the computers on the edge nodes are allocated increasing capabilities: the computer on node i, with $i \in [1..4]$, is allowed to use i out of the physical CPU cores of the server on which the experiments are run. To avoid mutual interference between serverless executors, each is statically allocated to a unique set of CPUs, the total number being greater than $\sum_{i=1}^{4} i$. In the following we only report the latency of "tagged" clients, i.e., one per node (see Fig. 14, again). The arrival rate of tasks at clients is distributed according to a Poisson r.v. with

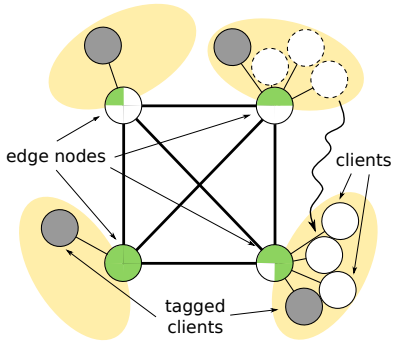

Figure 14: Network topology used in the small-scale testbed experiments. The number of slices per edge nodes indicates how many physical CPU cores have been reserved to the corresponding computer.

average 1/s. The tagged clients use a picture of size 640×480 pixels, while the other clients draw randomly their input from a set of pictures with size from 320×240 to 1280×768 pixels. The number of other clients, roaming from one edge node to another selected randomly, increases from 1 to 4 depending on the experiment. In these experiments it is $W_S = 4$.

Our proposed solution, called *Est* here, is compared to two alternatives: Legacy and RR. *Legacy* is a traditional solution where the clients request the execution on the computer located in their current point of attachment. On the other hand, *RR* is taken from our previous work [40]: it dispatches tasks based on a round-robin algorithm, weighted on the measured latency, without performing any estimation or prediction as in the currently proposed solution.

In Fig. 15 we show the 90th percentile of the delay of tagged clients. As can be seen, Round Robin (RR) performance is lowest at low traffic load. This is because it tries to use all the available computers evenly due to its round-robin policy, even though they have heterogeneous capabilities. On the contrary, at high loads, Legacy has poorest performance because for some clients their current point of attachment becomes too heavily loaded. The Est performance is intermediate between that of the two opposite approaches, which confirms that our solution is flexible in adapting to a heterogeneous environment and changing conditions, even though it does not have any *a priori* knowledge on the topology and capabilities of the edge nodes.

We further elaborate on this behavior with the help of Fig. 16, showing the average load of the computers at peak load. Legacy has an almost flat utilization: this is rather inefficient, because the load is evenly distributed across edge nodes but the capabilities are not. Note that a condition of unbalanced capabilities is far from being artificial: rather, in a real environment it is very likely that hardware and software on edge nodes, unlike their cloud counterparts, will be highly heterogeneous due to incremental deployment and fragmented ownership. On the other hand, both Est and RR achieve a load distribution that matches well the availability of resources.

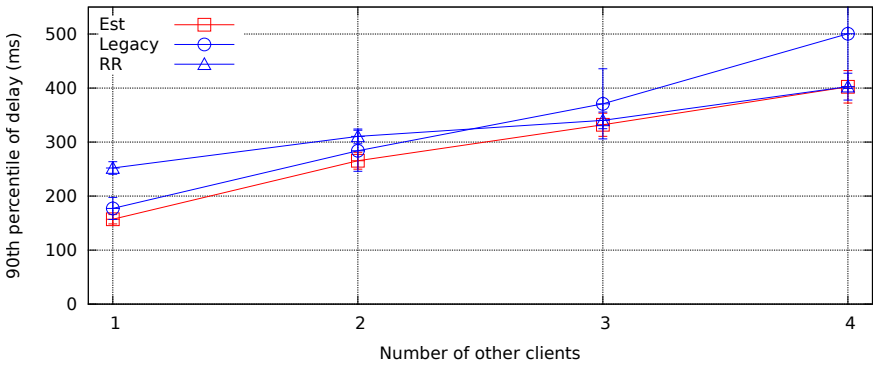

Figure 15: Small-scale experiment: 90th percentile of delay.

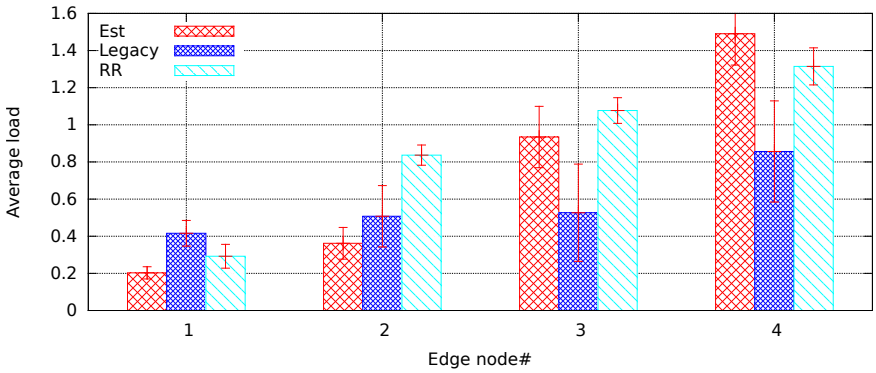

Figure 16: Small-scale experiment: Distribution of load across edge computers, with four other clients.

5.3. Sensitivity analysis

In this section we study the sensitivity of the proposed dispatching algorithm to variations of its internal parameters. We carry out this analysis using a $2^k r!$ analysis, which is an established methodology to identify the dominant factors in a statistically sound manner and, usually, plan experiments accordingly. Very briefly, for each of the k factors *potentially* affecting the performance, we identify two limit cases, identified by a $+$ and $-$ sign, respectively. Then, we run 2^k experiments in all possible combinations. We also repeat r times every experiment, without modifying the parameters. For every metric of interest, the $2^k r!$ method allows to derive the relative importance of every factor or combinations of factors, also quantifying the contributions in the same units as the performance index under study, within a given confidence interval. One basic assumption for this method to provide meaningful results is that the factors identified give additive contributions to the metric of interest (though the analysis may still be carried out with non-additive contributions with some

modifications). The interested reader may find additional information on this methodology in [38].

Table 5: Sensitivity $2^k r!$ analysis: factors.

Parameter		−	+
Number of other clients	A	1	4
T_τ	B	1	10
T_u	C	1	10
W_τ	D	50	200
W_u	E	50	200
Detect eyes	F	no	yes

The sensitivity analysis is carried out in the same topology as Sec. 5.2. The $k = 6$ factors identified are reported in Table 5 and they include internal parameters (W_τ, W_u, T_τ, T_u, see Sec. 3.2.3) as well as the following environmental parameters: the number of other clients, which is basically a measure of the computational/network load; and whether the serverless clients request only face detection or also the detection of eyes within every face found in the pictures. In the latter case all the computers offer two lambdas, one for face and the other for eyes detection, and the delay is the response time defined in Sec. 4.3. We performed the analysis in terms of the following metrics: delay, network throughput, load of the computers, and communication latency/processing time estimation error, defined as the absolute value of the difference between the *a priori* estimation and the *a posteriori* measurement done by the dispatcher for every lambda function. We run 10 independent replications for every combination of parameter, i.e. $r = 10$, yielding a total of $2^6 \cdot 10 = 640$ experiments executed.

Table 6: Sensitivity $2^k r!$ analysis: 90th percentile of delay (ms).

Effects	**Contribution**	**Conf. int.**
mean response (q0)	342	$(340, 343)$
effect of A (qA)	+75.6	$(73.7, 77.5)$
effect of F (qF)	+43.4	$(41.5, 45.3)$
other effects	*negligible*	
unexplained variations	9%	

In Table 6 we show the analysis in terms of the 90th percentile of delay, produced by the open source tool *factorial2kr* used[11]. Confidence intervals are computed with 95% level. From the table we see that the 90th percentile of delay is affected merely by environmental parameters A (number of other clients) and

[11] `https://github.com/ccicconetti/factorial2kr`

F (detection of face only vs. face+eyes). The sign of the qA and qF values also gives us the direction, which is as expected: the latency increases as the number of other clients increases from 1 to 4 (the average increase is 75.6 ms) and if we also include eyes detection (the average increase is 43.4 ms).

Table 7: Sensitivity $2^k r!$ analysis: median of processing time estimation error (ms).

Effects	**Contribution**	**Conf. int.**
mean response (q0)	22.2	$(21.8, 22.6)$
effect of A (qA)	$+13.4$	$(13.0, 13.8)$
effect of F (qF)	-19.7	$(-20.1, -19.3)$
joint effect of A&F (qAF)	-12.6	$(-12.9, -12.2)$
other effects	*negligible*	
unexplained variations	4%	

In Table 7 we report the analysis when considering the processing time estimation error $|\hat{p} - p|$. As for the 90th percentile of delay, only the environmental factors have non-negligible impact on the performance. However, there are two differences. First, the sign of qA and qF is opposite, which means that in this case the processing time estimation errors *increases* as the load increases, but it *decreases* if the clients also perform eyes detection. This is an effect due to the shorter duration of eyes detection compared to face detection, because when requesting the latter the clients only include as input the portion of picture corresponding to a single face. Second, a combination qAF appeared in the table: this means that 21% of the processing time error is due to the *joint* effect of the number of other clients and application of choice, which can be explained by the fact that having more serverless traffic also gives more measurements to the dispatcher, which can then be more precise in their estimations, especially with faster detection algorithms.

For all the metrics considered, we have verified that the results are valid in a statistical sense by visually inspecting the residual errors, whose residuals vs. predicted values and Q-Q normal plots are reported in the supplementary material.

5.4. Large-scale topology

We conclude the performance evaluation with the discussion of the results obtained with a large-scale realistic network topology, representing an edge system for mobile devices with AR applications in a urban scenario. We use simulated computers (see Sec. 4.2) because the size of the scenario did not allow us to run real AR lambda services on a single server. We use the open datasets with the recordings of human activity in the city of Milan (Italy) described in [41]. They cover a one-month duration and refer to a city landscape divided into a square grid of 10,000 cells. We assume that each cell contains a three-sector base station and that the base stations are then grouped into "pods" of three elements connected to a core network. As commonly found in operational networks, we

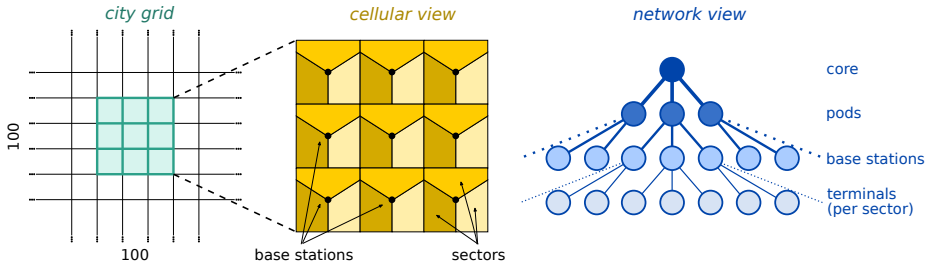

Figure 17: Mapping of the city grid from [41] into the network topology used for the experiments.

assign capacities so as to obtain a "fat-tree" network topology [42], with capacity halving when moving down one level from the root node, which has 1 Gb/s with 10 μs latency with its directly attached nodes. For the access network, i.e., the connection between the clients and their respective base station, we assume a 25 Mb/s capacity with 1 ms, which is a slightly optimistic estimate compared to actual findings in current 4G networks [43]. We illustrate in Fig. 17 the mapping of the city grid into our emulated network. In particular, the network view (right side) shows the connectivity graph, where the thickness of the edge is a qualitative indicator of the link speed: at the terminals level, one node represents all the user terminals in a sector; at the base stations level, one node represents a single (sectorized) base station, which acts as both a dispatcher and a computer, with two simulated cores dedicated to processing lambda functions; at the pods level, one node represents the collection of networking equipment creating a backhaul connection between the access and core networks; finally, the entire core network is collapsed into the root node of the tree.

We use a Monte Carlo approach for the experiments, described in the following. Inspired from the evaluation in [44], we extract from a random day in the dataset in [41] the Internet activity with a 10-minute granularity and use these data to set the load of each cell over time. Then, we perform a number of independent snapshots (or drops) of the system, selecting a random location of a group of 3×3 cells and dropping users in them with random arrival times with a rate proportional to the cell activity at the given (random) time of day. The duration of an AR session, consisting of consecutive lambda function calls every 33 ms (i.e., with a 30 fps frame rate), is extracted from a uniform r.v. between 30 s and 60 s. According to [45], we consider lambda requests/responses such that the application bandwidth is between 3 Mb/s and 10 Mb/s, with a 75 ms maximum tolerable total delay for the execution of every single lambda. In Table 8 we report the response times of the lambda function with some image sizes.

In the following we call our solution *Dist Est* and compare it to the following alternatives: Legacy, Centralized, and Dist Probe. Firstly, *Legacy*, like in Sec. 5.2, is a policy where each client requests lambda execution to its serving base station. Secondly, we call *Centralized* an approach where the tasks

Table 8: Response times of the lambda functions used in Sec. 5.4 with a simulated computed emulating an AR application.

Size (bytes)	**Comp. only (ms)**	**With network (ms)**
5000 bytes	9.0 ± 0.2	12.1 ± 0.4
10000 bytes	17.5 ± 0.2	24.1 ± 0.3
15000 bytes	25.9 ± 0.2	35.9 ± 0.5

are balanced in the root node of the network, mimicking what happens in a cloud computing oriented serverless solution. Lastly, *Dist Probe* is a distributed version of the Probe algorithm described in Sec. 5.1, inspired from [6].

In Fig. 18 we report the 90th percentile of the delay experienced by the fraction of users in the x-axis. For instance, with Dist Est we have a value of 40 ms at 0.7 users: this means that 70% of the users, at every random location and time of day, experienced a 90th percentile of delay that is smaller than 40 ms. Hence, for every policy, the fraction of dissatisfied users is the complement to 1 of the x-axis projection of the point where the corresponding curve meets 75 ms. It is worth noting that while in all the other experiments so far we have estimated the confidence intervals via running multiple runs, this is not possible with the Monte Carlo methodology used: every drop represents a possible state of the system at a given time, and any two drops may capture very different conditions (e.g., night-time vs. peak hours).

As can be seen, Legacy achieves poorest performance, consistently with the findings in [46]. This is because a static allocation of users to computers creates "hot spots" of requests at each base station whenever a high concentration of clients appears there. Instead, Dist Est achieves roughly the same performance as Centralized, which is remarkable even without considering that the latter has a slightly higher number of dissatisfied users. This is caused by the inefficiency of forcing all the transactions through the core network, which becomes especially relevant at high loads. Finally, in almost all cases, Dist Est has smaller delays than Dist Probe, though this gap lessens as the load increases, until the fraction of dissatisfied users for the two algorithms becomes the same.

In Fig. 19 we show the overall average per-drop network throughput, sorted on the y-axis values. This highlights how both Centralized and Dist Probe require a substantial higher network capacity than our proposed solution Dist Est, which is beaten on this ground only by Legacy, because the computation never leaves the base station. Therefore our proposed dispatching algorithm achieves a good trade-off between computational resources load balancing and protocol overhead.

6. Conclusions

In this paper we have proposed a novel architecture for the execution of stateless tasks in an edge computing system, whose execution passes through

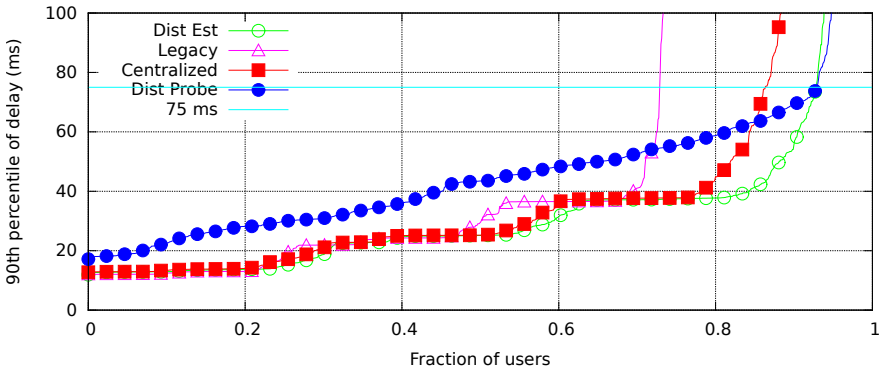

Figure 18: Large-scale experiment: Distribution of the 90th of delay.

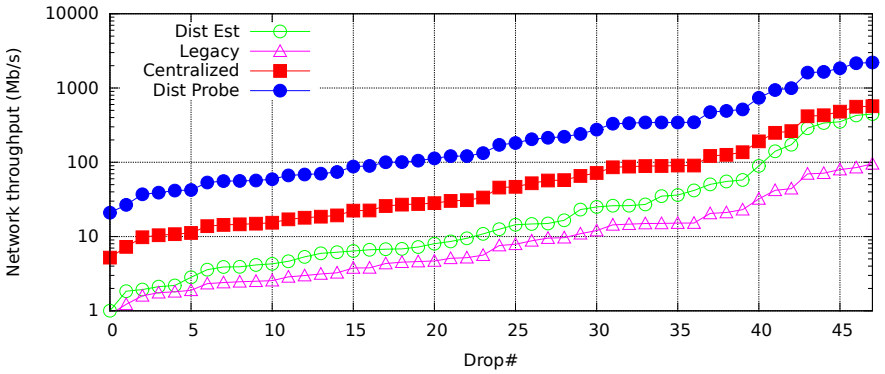

Figure 19: Large-scale experiment: Network throughput per drop.

nodes with dispatching capabilities, co-located with the points of attachment of the clients to the network. The proposed architecture is fully distributed: the dispatchers take online decisions purely based on a local estimation of the communication processing times, obtained using passive measurements.

Furthermore, to overcome the limitations of existing frameworks for the evaluation of edge computing systems, we have devised a novel approach that jointly exploits network emulation, process-based virtualization, and simulation of computation tasks to obtain accurate results in controlled repeatable conditions with reasonable computation effort. We have used our framework to validate the distributed algorithm to dispatch lambda functions in several scenarios. The results obtained in a large scale experiment show that the proposed distributed architecture achieves a 20% increase in user satisfaction compared to a static allocation of clients to computers, while matching the performance of two other approaches from the literature, which however require 1.7×-6.6× more network traffic.

As future work, we will work on the integration of our performance evaluation framework with edge computing platforms and communication stacks to take into account in a realistic manner the overhead and complexity introduced by, e.g., the ETSI MEC or the LTE Evolved Packet Core (EPC).

References

[1] M. Satyanarayanan, The Emergence of Edge Computing, Computer 50 (1) (2017) 30–39. doi:10.1109/MC.2017.9.

[2] C. Li, Y. Xue, J. Wang, W. Zhang, T. Li, Edge-Oriented Computing Paradigms, ACM Computing Surveys 51 (2) (2018) 1–34. doi:10.1145/3154815.
URL http://dl.acm.org/citation.cfm?doid=3186333.3154815

[3] OpenFog Consortium Architecture Working Group, OpenFog Reference Architecture for Fog Computing, OpenFogConsortium (February) (2017) 1–162. arXiv:OPFRA001.020817, doi:OPFRA001.020817.
URL https://www.openfogconsortium.org/wp-content/uploads/OpenFog{_}Reference{_}Architecture{_}2{_}09{_}17-FINAL.pdf

[4] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, D. Sabella, On Multi-Access Edge Computing: A Survey of the Emerging 5G Network Edge Cloud Architecture and Orchestration, IEEE Communications Surveys and Tutorials 19 (3) (2017) 1657–1681. doi:10.1109/COMST.2017.2705720.

[5] B. Varghese, R. Buyya, Next generation cloud computing: New trends and research directions, Future Generation Computer Systems 79 (2018) 849–861. arXiv:1707.07452, doi:10.1016/j.future.2017.09.020.
URL http://dx.doi.org/10.1016/j.future.2017.09.020

[6] H. Tan, Z. Han, X.-Y. Li, F. C. Lau, Online job dispatching and scheduling in edge-clouds, in: IEEE Conference on Computer Communications - INFOCOM, IEEE, 2017, pp. 1–9. doi:10.1109/INFOCOM.2017.8057116.
URL http://ieeexplore.ieee.org/document/8057116/

[7] A. Lertsinsrubtavee, A. Ali, C. Molina-Jimenez, A. Sathiaseelan, J. Crowcroft, Picasso: A lightweight edge computing platform, Proceedings of the 2017 IEEE 6th International Conference on Cloud Networking, CloudNet 2017doi:10.1109/CloudNet.2017.8071529.

[8] E. Saurez, K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, Incremental deployment and migration of geo-distributed situation awareness applications in the fog, 10th ACM International Conference on Distributed and Event-based Systems - DEBS'16 (2016) 258–269doi:10.1145/2933267.2933317.
URL http://dl.acm.org/citation.cfm?doid=2933267.2933317

[9] M. Król, I. Psaras, NFaaS: Named Function as a Service, in: Proceedings of the 4th ACM Conference on Information-Centric Networking - ICN '17, Vol. 11, ACM Press, New York, New York, USA, 2017, pp. 134–144. doi:10.1145/3125719.3125727.
URL https://doi.org/10.1145/3125719.3125727http://dl.acm.org/citation.cfm?doid=3125719.3125727

[10] A. Madhavapeddy, D. J. Scott, Unikernels: Rise of the Virtual Library Operating System, Queue 11 (11) (2013) 30:30—-30:44. doi:10.1145/2557963.2566628.
URL http://doi.acm.org/10.1145/2557963.2566628

[11] I.-D. Filip, F. Pop, C. Serbanescu, C. Choi, Microservices Scheduling Model Over Heterogeneous Cloud-Edge Environments As Support for IoT Applications, IEEE Internet of Things Journal 5 (4) (2018) 2672–2681. doi:10.1109/JIOT.2018.2792940.
URL https://ieeexplore.ieee.org/document/8255573/

[12] R. P. Singh, J. Grover, G. R. Murthy, Self organizing software defined edge controller in IoT infrastructure, in: 1st International Conference on Internet of Things and Machine Learning - IML'17, ACM Press, New York, New York, USA, 2017, pp. 1–7. doi:10.1145/3109761.3158390.
URL http://dl.acm.org/citation.cfm?doid=3109761.3158390

[13] S. Anand, N. Garg, A. Kumar, Resource Augmentation for Weighted Flow-time explained by Dual Fitting, in: Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, 2012, pp. 1228–1241. doi:10.1137/1.9781611973099.97.
URL http://dblp.uni-trier.de/db/conf/soda/soda2012.html{#}AnandGK12http://epubs.siam.org/doi/abs/10.1137/1.9781611973099.97

[14] S. Sthapit, J. Thompson, N. M. Robertson, J. Hopgood, Computational Load Balancing on the Edge in Absence of Cloud and Fog, IEEE Transactions on Mobile Computing (2018) 1–14doi:10.1109/TMC.2018.2863301.

[15] J. Edinger, D. Schafer, C. Krupitzer, V. Raychoudhury, C. Becker, Fault-avoidance strategies for context-aware schedulers in pervasive computing systems, 2017 IEEE International Conference on Pervasive Computing and Communications, PerCom 2017 (2017) 79–88doi:10.1109/PERCOM.2017.7917853.

[16] D. Schaefer, J. Edinger, M. Breitbach, C. Becker, Workload Partitioning and Task Migration to Reduce Response Times in Heterogeneous Computing Environments, in: 2018 27th International Conference on Computer Communication and Networks (ICCCN), Vol. 2018-July, IEEE, 2018, pp. 1–11. doi:10.1109/ICCCN.2018.8487326.
URL https://ieeexplore.ieee.org/document/8487326/

[17] P. Smet, B. Dhoedt, P. Simoens, Docker layer placement for on-demand provisioning of services on edge clouds, IEEE Transactions on Network and Service Management 4537 (c) (2018) 1–1. doi:10.1109/TNSM.2018.2844187.
URL https://ieeexplore.ieee.org/document/8372945/

[18] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, P. Merle, Elasticity in Cloud Computing: State of the Art and Research Challenges, IEEE Transactions on Services Computing 11 (2) (2018) 430–447. doi:10.1109/TSC.2017.2711009.
URL https://ieeexplore.ieee.org/document/7937885/

[19] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, R. Buyya, CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms, Softw. Pract. Exper. 41 (1) (2011) 23–50. doi:10.1002/spe.995.
URL http://dx.doi.org/10.1002/spe.995

[20] S. F. Piraghaj, A. V. Dastjerdi, R. N. Calheiros, R. Buyya, ContainerCloudSim: An environment for modeling and simulation of containers in cloud data centers, Software: Practice and Experience 47 (4) (2017) 505–521. doi:10.1002/spe.2422.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2422

[21] H. Gupta, A. VahidDastjerdi, S. K. Ghosh, R. Buyya, iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments, Software - Practice and Experience 47 (9) (2017) 1275–1296. arXiv:1606.02007, doi:10.1002/spe.2509.

[22] D. Fernández-Cerero, A. Fernández-Montes, A. Jakóbik, J. Kołodziej, M. Toro, SCORE: Simulator for cloud optimization of resources and energy consumption, Simulation Modelling Practice and Theory 82 (2018) 160–173. doi:10.1016/j.simpat.2018.01.004.
URL http://www.sciencedirect.com/science/article/pii/S1569190X18300030

[23] R. N. Calheiros, M. A. S. Netto, C. A. F. De Rose, R. Buyya, EMUSIM: an integrated emulation and simulation environment for modeling, evaluation, and validation of performance of Cloud computing applications, Software: Practice and Experience 43 (5) (2013) 595–612. doi:10.1002/spe.2124.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2124

[24] T. Qayyum, A. W. Malik, M. A. Khattak, O. Khalid, S. U. Khan, FogNetSim++: A Toolkit for Modeling and Simulation of Distributed Fog Environment, IEEE Access 6 (2018) 63570–63583. doi:10.1109/ACCESS.2018.2877696.

[25] S. Zhou, P. P. Netalkar, Y. Chang, Y. Xu, J. Chao, The MEC-Based Architecture Design for Low-Latency and Fast Hand-Off Vehicular Networking, 2018 IEEE 88th Vehicular Technology Conference (VTC-Fall) (2019) 1–7doi:10.1109/vtcfall.2018.8690790.

[26] S. K. Mohanty, G. Premsankar, M. D. Francesco, An evaluation of open source serverless computing frameworks, in: IEEE CloudCom, 2018.

[27] T. Lynn, P. Rosati, A. Lejeune, V. Emeakaroha, A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms, in: 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), IEEE, 2017, pp. 162–169. doi:10.1109/CloudCom.2017.15.
URL http://ieeexplore.ieee.org/document/8241104/

[28] R. Mayer, L. Graser, H. Gupta, E. Saurez, U. Ramachandran, EmuFog: Extensible and scalable emulation of large-scale fog computing infrastructures, in: 2017 IEEE Fog World Congress (FWC), IEEE, 2017, pp. 1–6. arXiv:1709.07563, doi:10.1109/FWC.2017.8368525.
URL http://arxiv.org/abs/1709.07563https://ieeexplore.ieee.org/document/8368525/

[29] C. Cicconetti, M. Conti, A. Passarella, Low-latency Distributed Computation Offloading for Pervasive Environments, in: 17th Annual IEEE International Conference on Pervasive Computing and Communications, 2019.

[30] A. Albanese, P. S. Crosta, C. Meani, P. Paglierani, GPU-accelerated Video Transcoding Unit for Multi-access Edge Computing Scenarios, Icn 2017 (c) (2017) 143–147.
URL http://www.academia.edu/download/52796922/icn{_}2017{_}full.pdf{#}page=155

[31] O. Ascigil, T. K. Phan, A. G. Tasiopoulos, V. Sourlas, I. Psaras, G. Pavlou, On Uncoordinated Service Placement in Edge-Clouds, in: IEEE International Conference on Cloud Computing Technology and Science, CloudCom, Vol. 2017-Decem, 2017, pp. 41–48. doi:10.1109/CloudCom.2017.46.

[32] R. Mailach, D. G. Down, Scheduling Jobs with Estimation Errors for Multi-server Systems, Proceedings of the 29th International Teletraffic Congress, ITC 2017 1 (2017) 10–18. doi:10.23919/ITC.2017.8064334.

[33] T. P. Pham, J. J. Durillo, T. Fahringer, Predicting Workflow Task Execution Time in the Cloud using A Two-Stage Machine Learning Approach, IEEE Transactions on Cloud Computing 7161 (c) (2017) 1–1. doi:10.1109/TCC.2017.2732344.
URL http://ieeexplore.ieee.org/document/8013738/

[34] S. Choochotkaew, H. Yamaguchi, T. Higashino, D. Schäfer, J. Edinger, C. Becker, Self-adaptive Resource Allocation for Continuous Task Offloading in Pervasive Computing, 2018 IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops 2018 (2018) 663–668doi:10.1109/PERCOMW.2018.8480400.

[35] R. Morabito, I. Farris, A. Iera, T. Taleb, Evaluating Performance of Containerized IoT Services for Clustered Devices at the Network Edge, IEEE Internet of Things Journal 4 (4) (2017) 1019–1030. doi:10.1109/JIOT.2017.2714638.

[36] S. Wang, X. Zhang, Y. Zhang, L. Wang, J. Yang, W. Wang, A Survey on Mobile Edge Networks: Convergence of Computing, Caching and Communications, IEEE Access 5 (2017) 6757–6779. arXiv:1703.10750, doi:10.1109/ACCESS.2017.2685434.

[37] E. Schiller, N. Nikaein, E. Kalogeiton, M. Gasparyan, T. Braun, CDS-MEC: NFV/SDN-based Application Management for MEC in 5G Systems, Computer Networks 135 (2018) 96–107. doi:10.1016/j.comnet.2018.02.013.

[38] A. M. Law, Simulation Modeling and Analysis, McGraw-Hill series in industrial engineering and management science, McGraw-Hill, 2007.
URL https://books.google.it/books?id=6cC{_}QgAACAAJ

[39] M. Breitbach, D. Sch, J. Edinger, C. Becker, Context-Aware Data and Task Placement in Edge Computing Environments, in: IEEE PerCom, 2019, pp. 272–281.

[40] C. Cicconetti, M. Conti, A. Passarella, An Architectural Framework for Serverless Edge Computing: Design and Emulation Tools, in: IEEE International Conference on Cloud Computing Technology and Science (CloudCom), IEEE, 2018, pp. 48–55. doi:10.1109/CloudCom2018.2018.00024.
URL https://ieeexplore.ieee.org/document/8590993/

[41] G. Barlacchi, M. De Nadai, R. Larcher, A. Casella, C. Chitic, G. Torrisi, F. Antonelli, A. Vespignani, A. Pentland, B. Lepri, A multi-source dataset of urban life in the city of Milan and the Province of Trentino, Scientific Data 2 (2015) 150055. doi:10.1038/sdata.2015.55.
URL http://www.nature.com/articles/sdata201555

[42] M. Al-Fares, A. Loukissas, A. Vahdat, A scalable, commodity data center network architecture, ACM SIGCOMM Computer Communication Review 38 (4) (2008) 63. arXiv:S0167739X10002554, doi:10.1145/1402946.1402967.
URL http://portal.acm.org/citation.cfm?doid=1402946.1402967

[43] N. Bui, J. Widmer, Data-Driven Evaluation of Anticipatory Networking in LTE Networks, in: 2017 29th International Teletraffic Congress (ITC 29),

Vol. 1, IEEE, 2017, pp. 46–54. doi:10.23919/ITC.2017.8064338.
URL http://ieeexplore.ieee.org/document/8064338/

[44] A. Ceselli, M. Fiore, M. Premoli, S. Secci, Optimized assignment patterns in Mobile Edge Cloud networks, Computers and Operations Research 0 (2018) 1–14. doi:10.1016/j.cor.2018.02.022.

[45] T. Braud, F. H. Bijarbooneh, D. Chatzopoulos, P. Hui, Future Networking Challenges: The Case of Mobile Augmented Reality, in: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), IEEE, 2017, pp. 1796–1807. doi:10.1109/ICDCS.2017.48.
URL http://ieeexplore.ieee.org/document/7980118/

[46] F. Malandrino, S. Kirkpatrick, C.-F. Chiasserini, How Close to the Edge?, Proceedings of the 2016 ACM Workshop on Cloud-Assisted Networking - CAN '16 (2016) 37–42arXiv:1611.08432, doi:10.1145/3010079.3010080.
URL http://dl.acm.org/citation.cfm?doid=3010079.3010080