Low-latency Distributed Computation Offloading for Pervasive Environments

Claudio Cicconetti
IIT, National Research Council
Pisa, Italy
c.cicconetti@iit.cnr.it

Marco Conti
IIT, National Research Council
Pisa, Italy
m.conti@iit.cnr.it

Andrea Passarella
IIT, National Research Council
Pisa, Italy
a.passarella@iit.cnr.it

***Abstract*—Future pervasive applications, like mobile augmented reality, have huge bandwidth and computation demands and very stringent delay constraints. Edge computing has been proposed to cope with such challenging requirements, since it shortens significantly the distance between the end users and the servers. On the other hand, serverless computing is emerging among cloud technologies to respond to the need of highly scalable event-driven execution of stateless tasks. In this paper, we investigate the convergence of the two to enable very low-latency execution of short-lived stateless tasks whose computation is offloaded from the user terminal to servers hosted by or close to edge devices in mobile pervasive environments. We realized a proof-of-concept implementation to delve into the specific issue of efficient dispatching of tasks in a distributed manner to achieve high scalability. We evaluated our proposed algorithm with experiments in a large-scale emulated network environment, showing that our solution achieves similar or better delay performance than a centralized solution, with far less network utilization.**

***Index Terms*—online job dispatching, serverless computing, computation offloading, edge computing**

I. Introduction

Nowadays there are several pervasive applications that are computationally intensive while having stringent delay requirements, but whose computations do not depend on the device's state. Examples include Augmented Reality (AR) and location-aware real-time information services. In AR the most challenging task, computationally speaking, is the analysis of the image and the superimposition of artificial elements or relevant information, which only depends on the last frame (or sequence) captured. In real-time information services the user is provided with the result of a joint analysis involving low-volume user data, such as location or sensor measurements, and high-volume external data/models. All such applications are suitable for delegation of the logic to a third party, which brings several advantages. First, an application can exceed the computational capabilities of the device on which it is running. Second, delegation extends the charge cycle of battery-operated mobile devices because the bulk of the computation is not done by the device itself. Currently, computation offloading is implemented via Mobile Cloud Computing (MCC), which has the further benefits of automatic installation/upgrade of applications, scalable resource allocation and flexible billing.

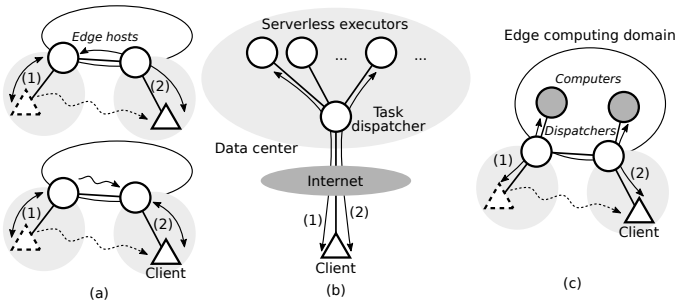

Fig. 1. Comparison of architectures for (a) edge computing, (b) serverless computing, and (c) proposed merge of the two.

Unfortunately, MCC creates a trade-off for latency-sensitive applications: offloading to a remote data center may incur longer response times due to sheer (network) distance between the application portion hosted by the user device and its cloud counterpart. To reduce latency, without dropping the main benefits of MCC, *edge computing* has emerged as a new paradigm where the functions that have been traditionally located in a remote data center are called back to some point nearer to the user [1], e.g., networking devices in the access network with spare or added computational capabilities. Edge computing technologies are being driven by vertical market segments, including: Internet of Things (IoT), for which a reference architecture has been published by the OpenFog Consortium [2]; the automotive and mobile network domains, which are of great interest to the telecom industry, which has recently standardized an inter-operable Multi-access Edge Computing (MEC) within European Telecommunications Standards Institute (ETSI) [3]. However, edge computing has its limitations when used in a pervasive environment with mobile devices: as shown in Fig. 1 (a) once an application has been provisioned on a given edge node, which is optimal for the current user position, if the user roams towards a different point of attachment the network must either accept increased latency due to sub-optimal routing (top part) or pay the cost of a migration of the application to another edge node (bottom part), which could also cause a service interruption.

To overcome this limitation we propose to merge the concepts of edge computing and *serverless computing*. Serverless computing [4] is a novel paradigm originating from cloud computing where clients request the execution of short-lived

978-1-5386-9148-9/19/$31.00 ©2019 IEEE

"light" jobs, e.g, a script in a given run-time environment, most often Python or Node.js. Such jobs are stateless, hence they do not require a full life-cycle management of the application nor the persistent allocation of resources on the remote server. This way, inherent scalability is achieved: ideally, no performance bottleneck exists as the number of monitoring events grow, as long as new locations for the execution of tasks are added. Existing solutions for serverless computing, such as Apache OpenWhisk[1] and Kubeless[2], adopt a logically centralized load balancer to dispatch the jobs to the servers, as show in Fig. 1 (b).

To the best of our knowledge, this is the first work where serverless computing is considered for the computational offloading of tasks in pervasive environments. In particular, we propose a solution for the execution of stateless jobs, called *lambda functions*, on edge nodes. Briefly, an edge computing domain is illustrated in Fig. 1 (c) and consists of: *computers*, which have computational capabilities to offer, because either they have spare resources, as often found in many networking devices, or they have been provisioned specifically for this purpose; *clients*, which are User Terminals (UTs) wishing to use said capabilities because it is impossible or inefficient for them to perform the computation directly; *dispatchers*, which are the entry points to the edge computing domain for clients. Clients perform Remote Procedure Calls (RPCs) on the dispatchers, which select the most suitable computer to run each function, forward to it the client request, then get back the result to the client. The RPC is short-lived: the input is contained in the request, the output in the response, and the call is closed immediately after completion to avoid the persistence of long-term states in the network. The architecture is fully distributed: dispatchers use only their local information to take decisions on which computer should execute the incoming function. Because of the ephemeral nature of lambda transactions, these decisions are not affected by the relocation of Virtual Machine (VM)/containers on computers, which happens far more sporadically. Such microservice-based approach is especially suitable to small devices acting as computers because the lambda execution bootstrapping overhead is negligible compared to that of loading a VM/container and the memory/storage footprint is reduced due to the absence of an application state on the computer. The proposed architecture achieves: i) *low latency*, since we cut away all the detours through the centralized decision point; ii) *scalability*, because the size of the decision problem at each dispatcher grows linearly with the number of computers; iii) *reliability*, as the failure of a dispatcher only affects the clients currently using it while the rest of the system runs without degradation.

The proposed solution has many research challenges associated. In this paper we focus specifically on the online algorithm used by the dispatchers to select the computer that will run a given function at a given time. As discussed in further details in Sec. II, state of the art solutions address on the one hand edge scenarios where tasks last much longer than in our case, and therefore tasks allocations need to be decided and changed much less frequently than in our reference environment. On the other hand, when tasks are "light-weight" as in our case, the target scenario is typically that of tasks scheduling in multi-core data centers, which is clearly a much more controlled and centralized environment than ours. We implemented a proof-of-concept of the proposed system to evaluate the online dispatching algorithm compared to a centralized approach, as commonly found in serverless computing in data centers, and a known online algorithm from the literature [5].

The remainder of this paper is organized as follows. In Sec. II we review the relevant state of the art. In Sec. III we describe the proposed system architecture and dispatching solution, which is then evaluated in Sec. IV. Conclusions are drawn in Sec. V.

II. State of the art

The contribution of this paper is two-fold: we propose an architecture for the convergence of the edge computing and serverless computing paradigms and we study the challenging problem of distributed dispatching of lambda functions to computers.

On the one hand, with regard to the architecture, in the scientific literature there are several proposals on how to realize computation offloading in edge computing. However, the vast majority are based on some form of lightweight orchestration, as compared to having a true "cloud" with VMs, by scaling down cloud-oriented paradigms to less powerful servers and faster dynamics. Examples include Picasso [6], from which we reuse the concept of providing the applications with an Application Programming Interface (API) whose routines are executed by the network in a manner transparent to clients, and *foglets* [7], which use containers for an easier and faster migration of functions based on situation-awareness schemes. Both studies put forward efficient ways to periodically tune the deployment of containers in edge servers, which is very relevant to our work but not addressed here.

In addition to generic architectures, such as what we propose in this paper, there are solutions tailored to specific scenarios. In [8] the authors exploit Information-Centric Networking (ICN) to realize a paradigm called *Named Function as a Service (NFaaS)*, where the functions are automatically distributed over ICN-enabled servers based on their utilization. Again, this could be a suitable complement to our present contribution, where we focus mostly on the short-term dispatch problem over an interval small enough that we can assume that the functions on the computers are stable. It is interesting to point out that to reduce the latency of setting up/tearing down the application, it is suggested that the servers employ *unikernels* [9], which are an extreme form of containerization. Finally, in [10] the authors propose an architecture to distribute jobs from IoT devices to a set of gateways by means of dynamic data plane manipulation: since all the client requests pass through the Software Defined Networking

[1]https://openwhisk.apache.org/

[2]https://kubeless.io/

(SDN) controller, the latter can estimate the arrival process and allocate new requests to servers accordingly. Unfortunately, such information is not available in a fully distributed approach such as ours.

On the other hand, from a high level perspective, the distributed dispatching problem can be described as follows, from the point of view of a given dispatcher. There are a number of lambda functions (or jobs) that will arrive over time and will have to be dispatched to a pool of computers, with the goal of minimizing their response time. The arrival process and the execution times are not known *a priori*. So far, this is a set-up for a classical multi-server scheduling problem, that has been extensively studied in the literature due to its huge importance in designing efficient schedulers in multi-core systems, both stand-alone and in grid/cloud environments. A known result is that no online algorithm[3] can have a bounded competitive ratio, see, e.g., [11]. In the same work the authors also propose a programmatic methodology to derive approximation algorithms that have a bounded competitive ratio in a speed augmentation model, i.e., by assuming that the online algorithm is given extra resources. One major difference with an edge computing scenario is that the multi-server scheduling problem assumes that the servers are used exclusively and that the policy for the execution of the jobs within each server is also under control. Both assumptions are false in our system: i) any computer can be assigned jobs by multiple non-communicating dispatchers, and ii) we cannot reasonably assume to have influence (or even insight!) on the scheduling within computers, which are highly heterogeneous (ranging from a Wireless Local Area Network (WLAN) routers to multi-core servers in a Mobile Network Operator (MNO) core network) and shared (e.g., a telco server may offer computational capabilities while also implementing Virtual Network Functions (VNFs)). Furthermore, different computers may have different communication latencies with the dispatcher, and they may vary over time since the network of an edge computing domain is expected to be also (well, actually mostly) used for Internet access.

A closer view is taken in [5], which in fact deals more specifically with edge computing. The authors propose an approximation algorithm to minimize the total weighted latency of the jobs, where the weight is assumed to be generically related to the delay-sensitiveness of the job. The algorithm is proved to be $\mathcal{O}(1/\epsilon)$-competitive in the $(1+\epsilon)$-augmented problem. The algorithm takes into account the communication latencies, which are assumed to be known for a given job, and it requires the processing time of every incoming job if executed on any given computer, which in general is not available. In our proof-of-concept we implemented the algorithm proposed in [5] when using emulated computers, which can provide the exact processing time of a job if no other arrives until its completion, as described in Sec. III-C. Comparison results with our proposed algorithm are reported in Sec. IV-B.

Finally, we mention the work of Edinger *et al.* [12], who envision a system where computation consumers ($\simeq$ *clients*) contact brokers ($\simeq$ *dispatchers*) that direct them to the most suitable computation producers ($\simeq$ *computers*), who are then contacted directly for the execution of so-called "tasklets" ($\simeq$ *lambdas*). Even though this environment looks similar to an edge computing scenario, there are two fundamental differences. Firstly, 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. Secondly, tasklets are assumed to be executed by end user devices, utterly unreliable. In fact, the most important scientific result of [12] is a scheduling algorithm that reduces the number of execution failures by estimating the reliability of producers. Such contribution is 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.

III. Serverless edge computing

In this section we describe the solution envisaged for the execution of stateless tasks, which is most suitable to applications like AR or real-time picture/video manipulation, with high computational demands but without a complex state or heavy storage usage. Then we delve into the design of the distributed algorithm for dispatching lambda functions to edge servers. We conclude the section with an overview of our proof-of-concept implementation.

A. Architecture

First of all, we briefly describe a typical serverless architecture, which is useful to better understand the novelty that we propose. As a reference implementation we use OpenWhisk, whose architecture is illustrated in Fig. 2. In addition to the *clients*, i.e., the applications that issue lambda functions, and *invokers*, i.e., the servers that actually run them, there are several ancillary components: NGINX[4] is an Hyper-Text Transfer Protocol (HTTP) load balancer that shapes client requests; Kafka[5] is a pub/sub platform that is employed to serialize the transactions with the invokers; CockroachDB[6] is employed to provide Authentication and Authorization (AA), to collect all possible actions and to store the transaction results. Additionally, the OpenWhisk controller plays a central role since it glues all the other back-end components together and is in charge of deciding which invoker to activate for a given function. As the controller receives a lambda execution request, after client's AA, it sets up a docker container on

[3] An *online* algorithm is one that takes a decision on a per-job basis and is not allowed to remain idle, in contrast to *offline* algorithms that have knowledge of past and future task arrivals, hence, may decide to delay a task even when the servers are idle to maximize the objective function.

[4] https://www.nginx.com/

[5] https://kafka.apache.org/

[6] https://www.cockroachlabs.com/

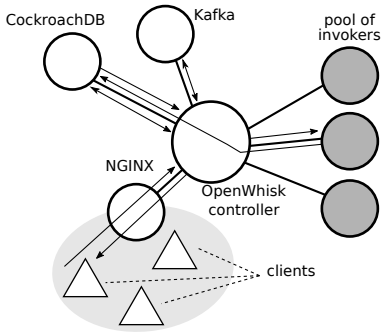

Fig. 2. Apache OpenWhisk architecture.

the selected invoker and runs the script/program downloaded from CockroachDB, whose results is eventually returned to the client.

OpenWhisk, like other serverless computing systems, cannot be used for computation offloading of delay-sensitive pervasive applications in edge networks, because in this context both the clients and the edge servers are distributed in a large geographical area and are interconnected with links having limited capacity and introducing significant delays, compared to the maximum tolerable latency.

The proposed architecture is illustrated in Fig. 3. We consider a generic Mobile Broadband Wireless Access (MBWA) where UTs connect to base stations, which are then interconnected though a core network of backhaul network devices. We assume that devices with computational capabilities, called *computers*, are co-located with the base stations, though not necessarily all of them, and with some of the core network devices. Such computers, in general, will have heterogeneous capabilities and may be equipped with hardware that is most suitable to execute a specific type of lambda functions, e.g., Graphics Processing Unit (GPU) for AR and video transcoding [13]. This is another difference with respect to the serverless computing world, where all the invokers are homogeneous and very often consist of specifically deployed VMs in the same facility. The base stations, which are the entry point to the network services for clients, act as *dispatchers*. In practice these base stations could be Long Term Evolution (LTE) e-NBs or WLAN access points or a mix of them.

We split the main system functions into two categories: offline and online. *Offline functions* are performed independently of lambda transactions and are expected to happen on a long time scale (minutes and above). *Online functions* are those associated to every lambda transaction, thus may happen at very short time scales (seconds and below). Since latency is a primary concern for pervasive real-time applications, we push as many functions as possible into the offline category: AA, set-up of the VM/containers on the computers, configuration of the dispatchers. Such offline functions are enabled by a logically centralized entity, called *controller*, which is represented as a server in the core network in Fig. 3. 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

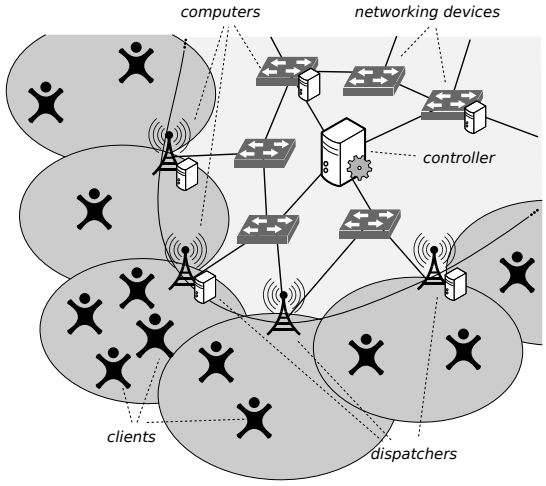

Fig. 3. Proposed distributed system architecture.

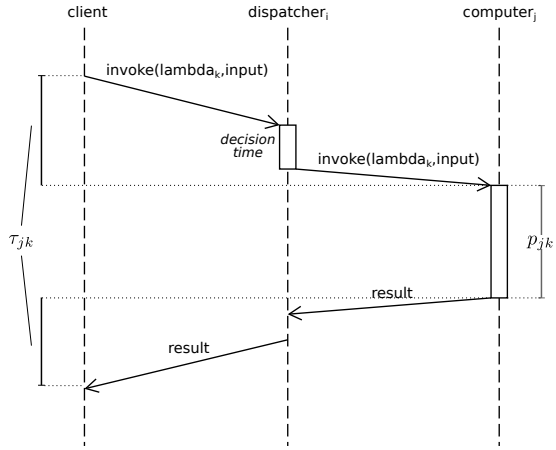

Fig. 4. Lambda request/response sequence.

by the computers. In the literature, the latter is referred to as "service placement" and some studies have already addressed this topic, e.g. [14], far from exhausting it, especially if heterogeneous hardware is considered. We do not address this issue here, and assume in the rest of the paper that in between consecutive re-organizations the set of VM/containers, hence lambda functions, in every computer is stable, thus lambdas can *immediately* be put into execution upon arrival from dispatchers, provided that there is sufficient hardware and software available: e.g., Central Processing Unit (CPU) and memory, pre-allocated workers and Operating System (OS)-related resources.

In Fig. 4 we show the sequence diagram of the only online function: the request of the activation of a lambda function from a client, also including the function input, its forwarding to the appropriate computer, and the final communication of the result to the issuing client. 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*.

B. Distributed lambda dispatching

We now present the algorithm for selecting the destination of a given lambda function $j \in \mathcal{L}$ ($\|\mathcal{L}\| = L$) at time t, provided that there is a set of $\mathcal{C}$ computers that can serve it[7], where $\|\mathcal{C}\| > 1$. We call $\delta_{jk}(t)$ the delay of job j if dispatched to computer k at time t. We can split $\delta_{jk}(t)$ into the following components: $\tau_{jk}(t)$, which is the time required for the transmission of the input from the client to the computer and for receiving the response on the way back, also including all queuing delays in intermediate transmission hops; and $p_{jk}(t)$, which is the time required from processing the lambda function j on computer k, which depends on its computational capabilities and other concurrent tasks sharing the resources with j until its completion. The components of the overall job delay are illustrated in Fig. 4. Ideally, the dispatching algorithm should select $\bar{k}$ such that:

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

where we have dropped the time t reference to simplify notation. This policy is well known in the literature under the name of Shortest Remaining Processing Time (SRPT) and is widely employed in multi-server schedulers because of its simplicity. In addition to having a bounded competitive ratio, it has been also shown to be more resilient than other sophisticated algorithms when the processing time is not certain [15], which is precisely our case because both τ_{jk} and p_{jk} cannot be known in advance. 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 $\hat{p}_{jk}$. Below we address the research challenge of estimating $\hat{\tau}_{jk}$ and $\hat{p}_{jk}$ in a manner that is i) *effective*, to emulate as closely as possible the behavior of an ideal SRPT scheduler, ii) *simple*, because the dispatcher has limited resources compared to, e.g., cloud servers in a data center, iii) *fast*, since we are targeting low-delay applications, therefore we cannot afford to linger too long on the decision of where to direct the lambda function, and iv) *subject to uncertainty*, for the dispatcher uses only local information, which is bound to become outdated quite fast in a highly dynamic pervasive environment.

1) Communication latency estimation: As far as $\hat{\tau}_{jk}$ is concerned, we propose a simple, yet effective, mechanism: when a computer is assigned a lambda function, it piggybacks the processing time into the response containing the result. This allows the dispatcher to sample the communication latency with every computer by simply keeping track of the overall time required for the job execution. Note that the dispatcher and computer do not need be synchronized since both time intervals are relative. With some simplifications, we can consider the communication latency as composed of two major components: a fixed offset, which depends only on the network topology and communication technologies used, and a variable quantity that is proportional to the amount of data transmitted. If we further assume that the lambda function output is either negligible compared to the input or proportional to it, then the dispatcher can collect for every computer a moving window of communication latency samples, obtained from the execution of *any* lambda function, and perform a simple linear regression to derive $\hat{\tau}_{jk}$ once job j arrives, hence its input size is known. More sophisticated approaches can be used without affecting the core of our contribution.

2) Processing time estimation: The estimation of the processing time $\hat{p}_{jk}$ 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 [16], where the authors propose a Machine Learning (ML)-based cloud task execution prediction framework. Furthermore, accurate prediction requires application- and scenario-specific details to achieve best accuracy. 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 and the internal details of computers (OS, scheduling policy, etc.).

Firstly, we assume that every computer is able to piggyback on the responses the current system load. This assumption is rather weak since every modern OS is able to provide effortlessly such information to its applications[8]

Secondly, we observe that in practice it is reasonable to expect an increasing relation between the processing time of a given lambda function with given input size and the load in the recent past: 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:

$$\langle \text{size}, \text{load} \rangle \rightarrow \text{processing time} \tag{2}$$

that can be used to extrapolate $\hat{p}_{jk}$.

In this work we take a simplistic approach and assume that p_{jk} is a linear function of both the lambda input size and the computer load. Under this assumption, estimation of the processing time can be done by finding the plane that best fits the population of samples collected. Since this fitting

[7]As a recap, this means that the controller, or any other orchestration function in the network, has configured all the VM/container/run-time environments necessary for the execution of that lambda function on all computers in $\mathcal{C}$ and that the dispatcher has been informed of such function placement before job j arrives.

[8]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; this is a mere implementation detail, though.

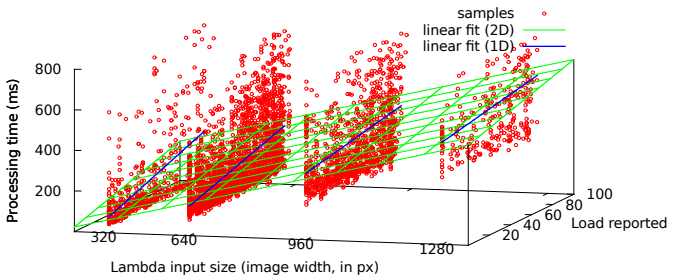

Fig. 5. Example of 2D and 1D linear fitting of processing data values in the dispatcher, for each computer k and lambda function j.

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. 5, 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. IV-A 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 confirms the above working assumption.

If the assumptions in this section do not apply to a specific scenario, for instance the execution time does not correlate to the input size, then another processing time estimation algorithm (e.g., one more sophisticated or that has white-box knowledge of the applications or computers) can be plugged in seamlessly when implementing the edge dispatching, without affecting the overall framework proposed.

3) Overall dispatching algorithm: To summarize, every time a lambda function j of size S_j is correctly executed by computer k, which reports load u_k and processing time p_{jk}, the dispatcher performs the following house-keeping 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;
4) add $\{u_k \rightarrow p_{jk}\}$ 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.

The final 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.:

$$\bar{k} = \arg\min_k \left\{ (\alpha_k^\tau + \beta_k^\tau S_j) + \left(\alpha^p_{S'jk} + \beta^p_{S'jk} u_k \right) \right\} \quad (3)$$

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 I, 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 I
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	–	$\mathcal{O}(C)$

C. Proof-of-concept implementation

To demonstrate the feasibility of the proposed architecture and evaluate the performance of the distributed lambda dispatching algorithm we implemented a proof-of-concept. We realized all the components involved, i.e., clients, dispatchers, computers, and controller. To simplify the discovery phase, we assumed that every computer and dispatcher registers itself to the controller at a known Uniform Resource Locator (URL). To achieve interoperability in a real deployment we envisage adopting standard APIs, such as those defined by the ETSI MEC [17]; such an opportunity will be investigated in a future work.

Execution of lambda functions has been implemented by means of REST interface methods using Google's gRPC[9]. Every *lambda request* contains the following fields: 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 A *lambda response* contains: a return code specifying what went wrong, if anything; the output; the 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

[9]http://grpc.io

they are attached, but integration with any other high-level programming language is possible since the gRPC library is platform and language independent.

We implemented two types of computers. Firstly, an *image manipulation computer* that performs face detection using the OpenCV library[10]. We selected image manipulation as an example application since the latter, together with AR, is a key application that can benefit from computational offloading in a pervasive environment. However we are not interested on the details and challenges associated to the specific use case and we refer the interested reader to, e.g., [18]. Secondly, an *emulated computer*, that performs arbitrary functions without really implementing any algorithm: rather, it just simulates internally the execution of the currently scheduled lambda functions to mimic the behavior of a multi-core multi-container edge server, where tasks are served according to a First Come First Serve (FCFS) non-preemptive policy. The simulated processing time depends on the input size and the actual model: in the experiments reported in the next section we have used a linear model where the processing time of a lambda function arriving on an empty computer is given by a constant offset plus the input size (in bytes) times a constant slope value. This type of 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 [5] because it can predict the completion time of jobs (if no other arrives meanwhile).

IV. Performance evaluation

In this section we evaluate the distributed lambda dispatching algorithm with our proof-of-concept implementation in two setups: a small-scale network of edge nodes, equipped with real face detection capabilities; a more realistic large-scale environment where we have used emulated computers.

Network emulation is done with Mininet[11], as proposed in EmuFog [19], which provides a wrapper for an easier configuration of edge/fog computing topologies. In all the experiments we have used $W_\tau = W_p = 100$ based on preliminary calibration experiments whose results are not reported here due to space limitations. The value of W_S depends on the specific experiment and is indicated in the respective sections below. For all these dispatching algorithm internal parameters we have found in the experiments performed that small variations have a negligible effect on the results. Further details on the evaluation tool are available in [20].

A. Small-scale experiments

In this set of experiments we arranged four edge nodes interconnected in a clique with 100 Mb/s links with 1 μs latency. This is representative, for example, of a local edge environment supporting a group of mobile nodes, deployed by a MEC operator through a set of edge gateways located close to each other. Each edge node hosts an image manipulation computer that can perform face detection via the execution of lambda functions as described in Sec. III-C. The computers are assigned different capabilities: the computer on node i, with $i \in [1..4]$, can use up to i CPU cores among those available in the server hosting the experiments. All clients connect to the edge nodes via links with 25 Mb/s capacity with 100 μs latency. For the convenience of evaluation we measure the latency of "tagged" clients only, one per edge node, that issue, on average, one lambda request per second with picture size 640x480 pixels according to a Poisson distribution. The number of other clients, requesting detection on pictures randomly drawn from a set of images from 320x240 to 1280x768 pixels and roaming from one edge node to another selected randomly, increases from 1 to 4. In these experiments it is $W_S = 4$. The response times for the different image sizes are reported in Table II (top section). The variance of the face detection response times is rather high due to the ML algorithm used in the OpenCV library for detection.

TABLE II
Response times of the lambda functions used in the experiments with an empty computer.

Size	Comp. only (ms)	With network (ms)
Face detection (real)		
320×240	43 ± 9	60 ± 10
640×480	101 ± 10	218 ± 26
960×720	181 ± 13	446 ± 47
1280×960	301 ± 13	744 ± 45
Augmented reality (emulated)		
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

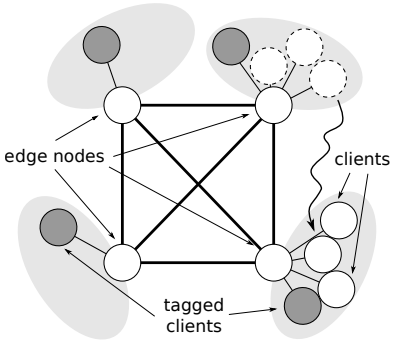

Fig. 6. Network topology used in the small-scale testbed experiments.

We compare the performance obtained with our proposed dispatching solution, called Est in the following, with two alternative approaches. First, a Round Robin (RR) algorithm, taken from our previous work [20], which classifies the computers based on lower vs. higher response time, then dispatches the incoming lambda request to the computers that are currently in the lower category. RR does not distinguish between communication and processing delays and does not use the load values reported by the computers. Second, we

[10]https://www.opencv.org/
[11]http://mininet.org/

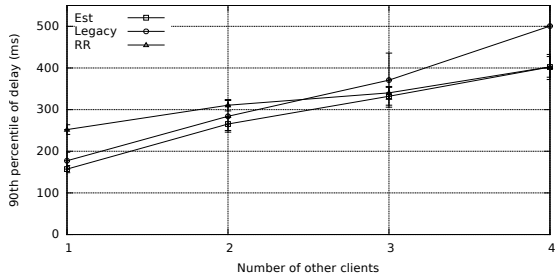

Fig. 7. Small-scale experiment: 90th percentile of delay.

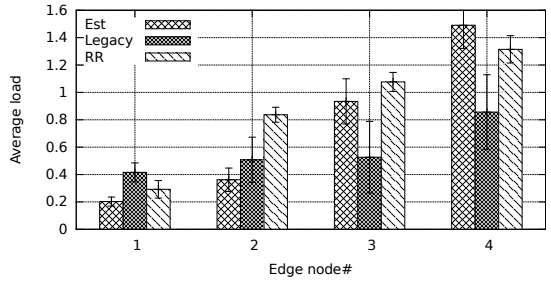

Fig. 8. Small-scale experiment: Distribution of load across edge computers, with four other clients.

consider a Legacy approach, where the clients simply request the execution to the closest computer, in number of hops.

Every experiment has been repeated 10 times: in the plots we report the average and 95% confidence interval over all the replications.

In Fig. 7 we show the 90th percentile of the delay of tagged clients. As can be seen, at low network loads RR performs worse that the others, because it strives to use evenly the available computers, which however have different capabilities. This behavior pays off at high loads, where, on the other hand, Legacy is penalized because it cannot cope well with "hot spots" of clients. In all cases the Est curve lies always below the others: our proposed approach can adapt well to mixed environments. Recall that this is achieved in a fully distributed manner and without providing Est with any *a priori* knowledge on the topology and capabilities of computers.

The reason is explained with the use of Fig. 8, which shows the average load of the computers at peak load. As can be seen, Legacy has an almost flat utilization, which is inefficient since the load is evenly distributed across edge nodes but the capabilities are not. On the other hand, both Est and RR distribute the load proportional to where there are more resources. For instance, even though computer 1 has lowest capabilities, it has non-negligible utilization with Est, which is thus able to harvest resources even from less powerful computers as necessary.

B. Large-scale topology

In this section we report the results obtained with a large-scale realistic network topology, where lambda functions are executed on emulated computers. The target application is AR on mobile devices in a dense urban environment. We use as reference real-world traces available as open datasets and described in [21], which include recordings of human activity in the city of Milan (Italy) for one month. The city landscape is divided into a square grid of 10,000 cells. We assume that each cell contains a base station serving users divided into three sectors. Base stations are grouped into sets of three elements, forming so-called pods, that are then connected to a common core. The resulting network topology is a fat-tree, commonly found in data-centers and operator core networks [22], where the network links between the root node and its children have 1 Gb/s with 10 μs latency, whereas capacity is halved in the tier below. The links connecting the clients to their respective base station have a 25 Mb/s capacity with 1 ms, which is a slightly optimistic estimate with respect to actual findings in current 4G networks [23]. The mapping of the city grid into an emulated network for the experiments in illustrated in Fig. 9. Each base station also acts as both a dispatcher and a computer, with two cores entirely dedicated to processing lambda functions.

For the experiments we used a Monte Carlo approach, which is widely employed for system-level performance evaluation of MBWA algorithms and protocols and is carried out as follows. Inspired from the evaluation in [24], from a random day in the dataset in [21] we extracted Internet activity with a 10-minute granularity, that is used to determine the cell load at every given time of day. Then, we perform a number of independent snapshots (or drops) of the system[12]. For each snapshot we select a random location of a group of 3x3 cells and a random time of day. Then, we drop users with random arrival times with a rate that is proportional to each cell activity at the given time. Each user establishes a session of AR with a duration randomly extracted between 30 s and 60 s, consisting of a stream of lambda function requests directed to the dispatcher co-located with the serving base station. During a session, consecutive lambda functions are issued every 33 ms, which corresponds to a frame rate of 30 fps. The size of lambda requests/responses is such to have bandwidth demands ranging from 3 Mb/s (lambda size 5000 bytes) to 10 Mb/s (lambda size 15000 bytes), which according to the authors in [25] is a reasonable compromise for good quality AR under realistic network constraints. The response times of the lambda function with some image sizes are reported in Table II (bottom section). We consider 75 ms as the maximum tolerable round-trip delay for the execution of a lambda function [25]. In the dispatchers we used a value of W_S such that lambda sizes are quantized every 1000 bytes.

We compare our proposed solution, called Dist Est (= distributed with processing estimation) below, to the following alternatives. First, a centralized approach where the dispatcher

[12]In the small-scale experiments we have measured and plotted confidence intervals through the execution of multiple independent replications of the very same scenario. With a Monte Carlo approach the notion of "independent replication" is a lot more blurred because 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). Therefore, rather than taking averages and reporting some measure of the variance, as customary with the method of independent replications, we report the full results obtained in all the drops using distributions.

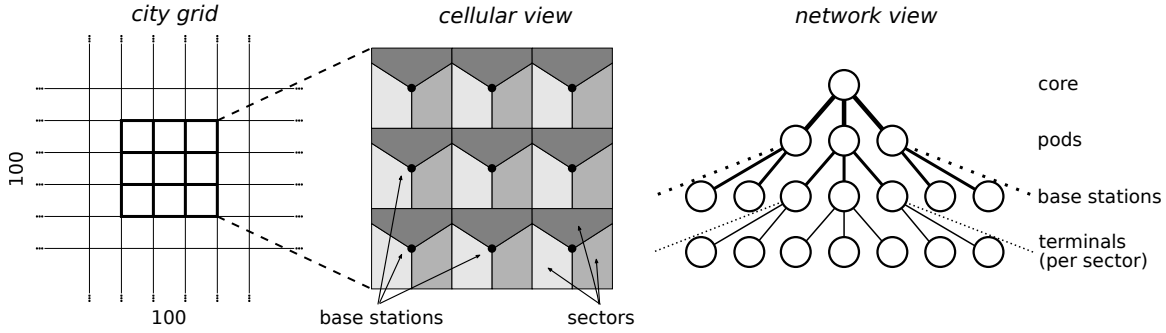

Fig. 9. Mapping of the city grid from [21] into the network topology used for the experiments.

is located in the root node of the topology tree, which mimics the behavior of a typical serverless solution, such as OpenWhisk. Second, an algorithm, called Dist Probe (= distributed with probing), inspired from [5], that is distributed but polls each computer to retrieve the execution time required, then selects the one that advertised the smallest value. Such an implementation is feasible since use emulated computers (see Sec. III-C) can simulate precisely the future evolution provided that no other jobs arrive. Third, like in Sec. IV-A, Legacy, where the clients request the execution of lambdas to their serving base station.

In Fig. 10 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. Therefore, we can compare these values to the 75 ms target and find the fraction of dissatisfied users as the complement to 1 of the x-axis projection of the point where each curve meets 75 ms. In this respect, Legacy achieves poorest performance: this is because a static allocation leads to under-utilization of the computational resources, which is consistent with the main finding in [26]. This is mainly due to the fact that with Legacy we create "hot spots" of requests at each base station whenever a high concentration of clients appears at that base station. Instead, Dist Est achieves roughly the same performance as Centralized, which however suffers from a slightly higher number of dissatisfied users because of the inconvenience of forcing all the transactions to pass through the root node, i.e., the core network, which is more prominent at high loads. Finally, Dist Est enjoys smaller delays than Dist Probe is almost all cases, but the performance gap lessens as the load increases. In particular, the fraction of dissatisfied users for the two algorithms is the same, though Dist Probe incurs a much more exorbitant cost in terms of network consumption, as discussed below.

This effect of excessive resort to network, which negatively affects the delay performance, is shown directly in Fig. 11, which reports the overall average per-drop network throughput, sorted on the y-axis values. Clearly, Legacy has minimum network consumption because the computation offloading traffic never leaves the base station. However, as can be seen, the network overhead caused by Dist Est with respect to Legacy is

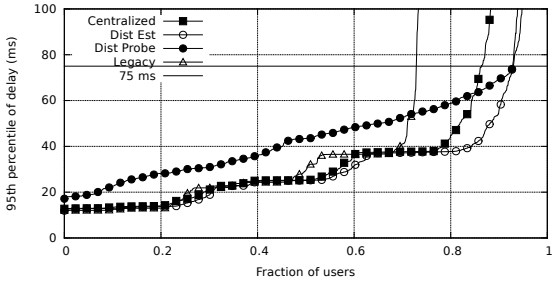

Fig. 10. Large-scale experiment: Distribution of the 90th of delay.

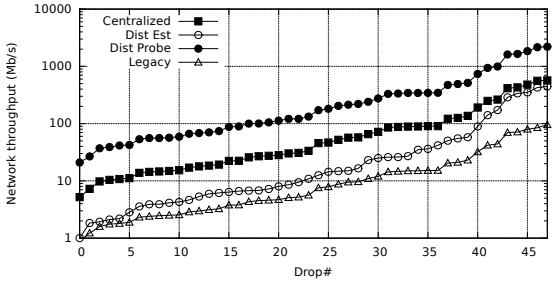

Fig. 11. Large-scale experiment: Network throughput per drop.

negligible compared to that of both Centralized and Dist Probe. This confirms that our proposed dispatching algorithm is able to achieve a good trade-off between wise utilization of the computational resources available and protocol/architecture complexity to achieve this goal, which ultimately benefits application latency in addition to traffic exchange.

V. Conclusions

In this paper we have proposed a system to offload pervasive applications with stringent delay requirements as stateless functions on edge nodes with available computational capabilities, called computers. The execution of functions passes through edge nodes with dispatching capabilities, which are ideally located as close as possible to the final users. The proposed architecture is highly scalable as the number of both the clients and the computers grow. We have implemented a proof-of-concept of the proposed solution and, through a set of experiments performed in an emulated network environment, we have shown that such a distributed architecture performs much better than statically allocating clients to computers and the same as or better than both a centralized approach and a distributed comparison solution from the literature.

REFERENCES

[1] Chao Li, Yushu Xue, Jing Wang, Weigong Zhang, and Tao Li. Edge-Oriented Computing Paradigms. *ACM Computing Surveys*, 51(2):1–34, apr 2018.

[2] OpenFog Consortium Architecture Working Group. OpenFog Reference Architecture for Fog Computing. *OpenFogConsortium*, (February):1–162, 2017.

[3] Tarik Taleb, Konstantinos Samdanis, Badr Mada, Hannu Flinck, Sunny Dutta, and Dario 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):1657–1681, 2017.

[4] Blesson Varghese and Rajkumar Buyya. Next generation cloud computing: New trends and research directions. *Future Generation Computer Systems*, 79:849–861, 2018.

[5] Haisheng Tan, Zhenhua Han, Xiang Yang Li, and Francis C.M. Lau. Online job dispatching and scheduling in edge-clouds. *Proc. of IEEE INFOCOM*, 2017.

[6] Adisorn Lertsinsrubtavee, Anwaar Ali, Carlos Molina-Jimenez, Arjuna Sathiaseelan, and Jon Crowcroft. Picasso: A lightweight edge computing platform. *Proc. of IEEE CloudNet*, 2017.

[7] Enrique Saurez, Kirak Hong, Dave Lillethun, Umakishore Ramachandran, and Beate Ottenwälder. Incremental deployment and migration of geo-distributed situation awareness applications in the fog. *Proc. of ACM DEBS*, pages 258–269, 2016.

[8] Michał Król and Ioannis Psaras. NFaaS: Named Function as a Service. *Proc. of ACM ICN*, pages 134–144, 2017.

[9] Anil Madhavapeddy and David J Scott. Unikernels: Rise of the Virtual Library Operating System. *Queue*, 11(11):30:30—-30:44, dec 2013.

[10] Rhishi Pratap Singh, Jitender Grover, and Garimella Rama Murthy. Self organizing software defined edge controller in IoT infrastructure. *Proc. of IML*, 2017.

[11] S. Anand, Naveen Garg, and Amit Kumar. Resource Augmentation for Weighted Flow-time explained by Dual Fitting. *Proc. of ACM-SIAM Symposium on Discrete Algorithms*, pages 1228–1241, 2012.

[12] Janick Edinger, Dominik Sch, Christian Krupitzer, Vaskar Raychoudhury, and Christian Becker. Fault-Avoidance Strategies for Context-Aware Schedulers in Pervasive Computing Systems. *Proc. of IEEE PerCom*, 2017.

[13] A Albanese, P S Crosta, C Meani, and P Paglierani. GPU-accelerated Video Transcoding Unit for Multi-access Edge Computing Scenarios. *Proc. of ACM ICN*, pages 143–147, 2017.

[14] Onur Ascigil, Truong Khoa Phan, Argyrios G. Tasiopoulos, Vasilis Sourlas, Ioannis Psaras, and George Pavlou. On Uncoordinated Service Placement in Edge-Clouds. *Proc. of IEEE CloudCom*, pages 41–48, 2017.

[15] Rachel Mailach and Douglas G. Down. Scheduling Jobs with Estimation Errors for Multi-server Systems. *Proc. of International Teletraffic Congress*, pages 10–18, 2017.

[16] Thanh Phuong Pham, Juan J. Durillo, and Thomas Fahringer. Predicting Workflow Task Execution Time in the Cloud using A Two-Stage Machine Learning Approach. *IEEE Transactions on Cloud Computing*, 7161(c):1–1, 2017.

[17] E. Schiller, N. Nikaein, E. Kalogeiton, M. Gasparyan, and T. Braun CDS-MEC: NFV/SDN-based Application Management for MEC in 5G Systems. *Computer Networks*, 135:96–107, 2018.

[18] Shuo Wang, Xing Zhang, Yan Zhang, Lin Wang, Juwo Yang, and Wenbo Wang. A Survey on Mobile Edge Networks: Convergence of Computing, Caching and Communications. *IEEE Access*, 5:6757–6779, 2017.

[19] Ruben Mayer, Leon Graser, Harshit Gupta, Enrique Saurez, and Umakishore Ramachandran. EmuFog: Extensible and scalable emulation of large-scale fog computing infrastructures. *Proc. of IEEE Fog World Congress*, 2017.

[20] Claudio Cicconetti, Marco Conti, and Andrea Passarella. An Architectural Framework for Serverless Edge Computing: Design and Emulation Tools. *Proc. of IEEE CloudCom*, 2018.

[21] Gianni Barlacchi, Marco De Nadai, Roberto Larcher, Antonio Casella, Cristiana Chitic, Giovanni Torrisi, Fabrizio Antonelli, Alessandro Vespignani, Alex Pentland, and Bruno Lepri. A multi-source dataset of urban life in the city of Milan and the Province of Trentino. *Scientific Data*, 2:150055, oct 2015.

[22] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM Computer Communication Review*, 38(4):63, 2008.

[23] Nicola Bui and Joerg Widmer. Data-Driven Evaluation of Anticipatory Networking in LTE Networks. *Proc. of International Teletraffic Congress*, pages 46–54, 2017.

[24] Alberto Ceselli, Marco Fiore, Marco Premoli, and Stefano Secci. Optimized assignment patterns in Mobile Edge Cloud networks. *Computers and Operations Research*, 0:1–14, 2018.

[25] Tristan Braud, Farshid Hassani Bijarbooneh, Dimitris Chatzopoulos, and Pan Hui. Future Networking Challenges: The Case of Mobile Augmented Reality. *Proc. of IEEE ICDCS*, pages 1796–1807, 2017.

[26] Francesco Malandrino, Scott Kirkpatrick, and Carla-Fabiana Chiasserini. How Close to the Edge? *Proc. of ACM CAN*, pages 37–42, 2016.