Computer Networks 172 (2020) 107184

Contents lists available at ScienceDirect

Computer Networks

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

Uncoordinated access to serverless computing in MEC systems for IoT

Claudio Cicconetti*, Marco Conti, Andrea Passarella

IIT, National Research Council, Pisa, Italy

ARTICLE INFO

Keywords:
Online job dispatching
Serverless computing
Computation offloading
Performance evaluation
Distributed cloud
Internet of Things
Mobile Edge Computing

ABSTRACT

Edge computing is a promising solution to enable low-latency Internet of Things (IoT) applications, by shifting computation from remote data centers to local devices, less powerful but closer to the end user devices. However, this creates the challenge on how to best assign clients to edge nodes offering compute capabilities. So far, two antithetical architectures are proposed: centralized resource orchestration or distributed overlay. In this work we explore a third way, called uncoordinated access, which consists in letting every device exploring multiple opportunities, to opportunistically embrace the heterogeneity of network and load conditions towards diverse edge nodes. In particular, our contribution is intended for emerging serverless IoT applications, which do not have a state on the edge nodes executing tasks. We model the proposed system as a set of M/M/1 queues and show that it achieves a smaller delay than single edge node allocation. Furthermore, we compare uncoordinated access with state-of-the-art centralized and distributed alternatives in testbed experiments under more realistic conditions. Based on the results, our proposed approach, which requires a tiny fraction of the complexity of the alternatives in both the device and network components, is very effective in using the network resources, while incurring only a small penalty in terms of increased compute load and high percentiles of delay.

1. Introduction

Nowadays *edge computing* is a trending architecture where applications on user devices are provided with computational capabilities made available in the access networks. Compared with traditional Mobile Cloud Computing (MCC), edge systems enjoy lower latencies and reduced Internet traffic. These advantages make them desirable in several vertical market segments, including mobile Augmented Reality (AR)/Virtual Reality (VR) [1], connected car [2], and IoT [3]

Meanwhile, a new paradigm, called *serverless computing* or Function as a Service (FaaS), is also revolutionizing IoT frameworks [4]. In serverless computing, processing is offloaded from the user device by means of tasks similar to remote function calls, often called *lambda functions*, which are processed by remote executors in a stateless manner [5]. Serverless computing was born as a cloud computing technology to allow an easier up/down scaling of the executors in a data center since there is no server-side state to be handled. However, this paradigm fits very well many IoT applications that natively consist of event-driven or periodic execution of processing jobs on data acquired in real-time for monitoring purposes [6].

In this paper we consider a system for IoT applications that combines the advantages of edge systems and serverless computing. Our target scenario is illustrated by means of the example in Fig. 1, which shows an edge domain consisting of: (i) access points, which provide the client devices with access to the edge network; (ii) lambda executors, co-located with network devices, which are equipped with spare/extra compute capabilities to respond to function execution requests from the clients; (iii) clients, which offload their computation by means of lambda requests towards the executors; (iv) a logically centralized entity, indicated as controller / orchestrator, which manages the lifecycle of the lambda images on the executors and dispatches the lambda requests from clients. In the literature there are two alternative approaches for this scenario, which will be analyzed in Section 3.2: *centralized*, where the association between clients and executors is decided by the orchestrator, and *distributed*, where the edge nodes cooperate to dispatch lambda functions from clients to executors without a central coordination. Both approaches require that a network-wide infrastructure is created and maintained, which may incur a significant overhead and prove inflexible to fast changing conditions, especially if the capabilities of the lambda executors are limited, which is a use case of interest for IoT applications on low-power gateways in the network.

Therefore, to overcome the limitations of these existing approaches, we propose to provide the clients with **uncoordinated access**: a light orchestration assigns every client a pool of possible lambda executors, but the final choice on where to send each and every request is made by the client. We discuss this solution in Section 3, where we also propose a practical decision mechanism based on probing of the response time from different executors, which is simple enough to be implemented even in IoT devices with very limited computation resources on board.

* Corresponding author.
E-mail addresses: c.cicconetti@iit.cnr.it (C. Cicconetti), m.conti@iit.cnr.it (M. Conti), a.passarella@iit.cnr.it (A. Passarella).

https://doi.org/10.1016/j.comnet.2020.107184
Received 14 October 2019; Received in revised form 24 January 2020; Accepted 27 February 2020
Available online 28 February 2020

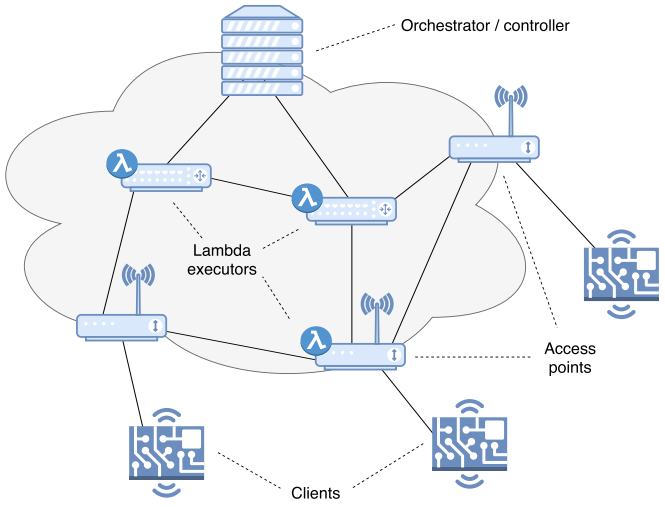

Fig. 1. Target scenario.

Furthermore, we show that our proposed solution is compatible with the European Telecommunications Standards Institute (ETSI) Multi-access Edge Computing (MEC) standard [7], which is attracting a growing interest from the edge computing industry, especially in the mobile telco domain. We provide the reader with a tutorial introduction to the standard in Section 2.2.

Furthermore, in Section 4 we model the proposed system under simplifying assumptions, which allows us to perform a numerical analysis of our proposed solution in the same section. Finally, in Section 5 we validate the conclusions obtained via analysis of experimental results obtained with a proof-of-concept implementation. Experiments are carried out in an emulated network, configured with realistic topology and traffic conditions, also comparing our uncoordinated access to centralized and distributed solutions from the literature.

2. State of the art

The goal of this section is two-fold. On the one hand, in Section 2.1 we provide the reader with an overview of the recent studies in the scientific literature that are most relevant to this work We note that, to the best of our knowledge, there are no works in the literature that address specifically the topic of serverless computing in edge systems for IoT; thus, we survey a selection of works that, in our opinion, provide an adequate technical background or ancillary solutions *in preparation of addressing* the challenges ahead. On the other hand, in Section 2.2 we introduce the ETSI MEC standard, which is relevant in its possible role as a leading technology to deploy interoperable edge systems and applications.

2.1. Distributed computing in edge systems

The amount of literature that could be ascribed to IoT is titanic. After the outburst of works on Wireless Sensor Networks (WSNs) more than 20 years ago, our research community has produced architectures, protocols, and algorithms for all possible requirements, some of which have made it into standards and products in the market. However, conclusive solutions have yet to be found regarding some crucial aspects that still hinder the full potential of mass applications to be unlocked, which is expected to pass through edge systems thanks to the advantages they offer compared to both on-device execution and pure cloud offloading, as already discussed in the introduction. These aspects include scalable and sustainable strategies for the operation and continuous optimization of resources under realistic assumptions, for which we illustrate the recent state of the art in the following. The interested reader may find further sources of inspiration in the recent survey papers [8,9].

The authors in [10] focus on the server-side implementation challenge if having multiple services on an edge node with compute capabilities, requiring isolation and low overhead, especially lower than that imposed by full-fledged virtualization systems intended for high-end servers. To this aim, they propose to use WebAssembly, which is a binary instruction format intended for applications to be executed with native speed within web browser, but could equally be used as a form of extremely down-scaled virtualization for IoT services, with similar goals as Unikernels [11]. An even more further looking solution is proposed in [12], where the micro-services are assumed to be dynamically distributed and executed based on peer-to-peer monetary incentives, which is a direction already pursued in the market, e.g., in the Golem network project https://golem.network/, though in the context of High Performance Computing (HPC). In any case, our work builds on top of any such approach that allows edge nodes to provide FaaS micro-services that respond to requests from clients: we aim at optimizing their access in the short-term (seconds to minutes), while long-term optimizations will be done regularly as part of the system's house-keeping activities.

Fault tolerance is the subject of some works, including [13], where distributed computing is realized by means of so-called *tasklets*. The underlying assumption there is that executors are inherently error-prone, because they are hosted on devices owned by (cooperative) end users. While this assumption may not apply to typical IoT scenarios, where failure of an edge node is expected to be a sporadic event, we note that our proposed uncoordinated serverless access goes exactly into the same direction, since it embeds reliability by using a pool of executors rather than a single one. Another problem that has attracted some interest recently is deciding on the user device whether a given task should be offloaded to edge/fog/cloud nodes or it would be better executed on the local compute resources. As in [14], this problem usually creates trade-offs between execution time and energy consumption, which fits very well the use cases where the clients are smart phones. On the other hand, a basic assumption of this work is that the IoT user devices have very limited computation capabilities for taking sophisticated decisions, and even more so for executing tasks by themselves. However, we inspire from that work for the definition of the mathematical system model in Section 4. An alternative solution has been also proposed in [15] for the same problem, where a near-optimal decision algorithm based on Q-learning is proposed.

Finally, we cite here the two alternative architectures mentioned briefly in the introduction, which will be studied in more details in Section 3.2: centralized vs. distributed. A centralized solution, where a single logical entity implements load-balancing on the client requests towards a pool of executors, is the standard approach in all cloud-based serverless environments, which have been evaluated, e.g., in [16] (open source) and [17] (commercial). On the other hand, in our previous work [18], we have proposed to distribute load balancing on the edge nodes themselves to overcome the limitations of a centralized structure in an irregular edge network. However, our previous solution was not intended for IoT scenarios, where both edge nodes and clients may have limited capabilities. In Section 5 we compare in a large-scale scenario the access scheme proposed in this paper to both such approaches.

2.2. ETSI MEC

The MEC industry study group was founded in ETSI in 2014[1] to create an open environment for the deployment of interoperable applications from all the actors in the edge ecosystem: vendors, service providers, third parties. As a matter of fact, most scientific works focus on specific aspects of the standard. In [19] the authors show how a real-time video streaming application may benefit form radio-level

[1] The original name of the committee was *Mobile* Edge Computing, later changed to *Multi-access* Edge Computing in accordance with the paradigm shift towards a technology-agnostic set of specifications, intended for not only mobile wireless networks.

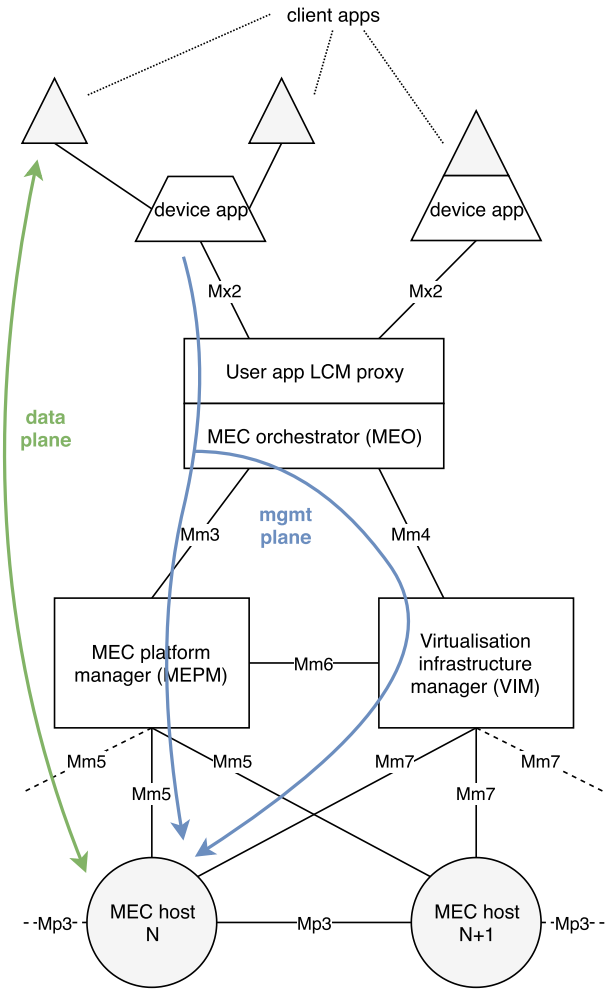

Fig. 2. Simplified blueprint of the ETSI MEC reference architecture.

information provided through the ETSI MEC interfaces, thus allowing a service provider to improve the Quality of Experience (QoE) of its users through the use of open interfaces (today the only option would be to sign a contract with every mobile network operator, then use different proprietary interfaces to gather information from each). In the core network of a mobile operator Software Defined Networking (SDN) and Network Function Virtualization (NFV) are the state-of-the-art solutions for the deployment and operation of services; in [20] the authors explore their relationship with ETSI MEC, which is a key aspect in a production network. The same problem is also addressed in [21], with a specific focus on redirecting traffic from a mobile user to its edge node in a transparent manner during roaming. However, it is yet to be understood whether SDN/NFV are also relevant for IoT systems where the devices for both connectivity and computation are expected to be more heterogeneous and with limited capabilities.

In the following we provide a short introduction to the standard, with a focus on the aspects that are more relevant to this work. In Fig. 2 we show the reference architecture of ETSI MEC, as of release 2.1.2 (in draft at the time of writing), with some simplifications regarding interfaces and components that are not relevant to the discussion in this paper. The interested reader is referred to [22], which provides a complete overview of the standard, or directly to the ETSI MEC specifications, which are all available to the public from the group website https://www.etsi.org/committee/1425-mec.

In the upper part of the figure we show the client and device apps. The *client app* is ETSI MEC agnostic and it interacts on the data plane with a *user app* that physically resides on a MEC host. The interface between the client and user app is application-dependent; generally, serverless computing is carried out by means of micro-services, hence the client only needs to know the Uniform Resource Locator (URL) or end-point of the lambda executor to offload computation tasks to it.

On the other hand, the *device app* is an ETSI MEC aware component of the application running on the user device, which interacts with the MEC platform on the management plane through the User Application Life Cycle Management (LCM) Proxy using the `Mx2` Application Programming Interface (API). The latter, as all the other ETSI MEC interfaces, is a vendor-neutral RESTful interface, whose commands and data structures are specified to facilitate interoperability between application and platform software, intended to be developed by different players in the ecosystem. The workflow expected from an application wishing to use an ETSI MEC service is the following:

1. the client app invokes computation offloading via a proprietary interface on its device app; note that the client and device app may reside in different devices, e.g., in an IoT system the client app may be in the smart object and the device app on a concentrator or gateway;
2. the device app checks the availability of the application requested and initiates the creation of an application context;
3. the MEC Orchestrator (MEO) checks the availability of resources and, if the new application is accepted, it allocates the necessary resources via the MEC Platform Manager (MEPM) and Virtualization Infrastructure Manager (VIM) using the `Mm3` and `Mm4` interfaces, respectively; the algorithms and criteria used by the MEO are voluntarily left open by the standard to foster market differentiation;
4. these requests, in turn, reflects on the MEC hosts via the `Mm5` and `Mm7` interfaces, respectively for computation and connectivity resources;
5. once this flow on the management plane is completed, the device app is notified on the `Mx2` interfaces and the client and user app can start data plane interactions.

At any time the MEO can change the MEC host serving the client app for optimization reasons by means of a push notification to the device app, e.g., if the mobile device roams to another area of the wireless network or if the computation/network conditions change due to other applications. With non-serverless applications, this also incurs a state migrations, which in general is a complex and costly operation. In Section 3.4 we will describe how to implement serverless uncoordinated access in ETSI MEC.

3. Uncoordinated serverless access

In this section we describe our proposed architecture for uncoordinated access of IoT clients to serverless micro-services in an edge system. We start by defining the key requirements of IoT architectures, in general, in Section 3.1, based on which we propose our so-called *uncoordinated access* in Section 3.2. We then illustrate in Section 3.3 a simple stateless algorithm that can be used by the clients in this architecture, intended as a baseline for constrained devices. Depending on the availability of extra computational resources on the client devices, one can think of more sophisticated solutions, which are the subject of our current investigations. We believe that our contribution is general enough to be suitable to several edge technologies and target deployments; to confirm our statement, in Section 3.4 we show how to realize the framework with the ETSI MEC.

3.1. Requirements

Before delving into the illustration of our contribution, we elaborate a moment on the following four **fundamental requirements** that *any* architecture should meet to be an effective solution in our context:

A. *It should be easy to implement on the user side:* IoT devices often have very limited CPU/memory capabilities.
B. *It should be lean on edge compute resources:* in an IoT scenario the network infrastructure usually consists of WiFi Access Points (APs) or other System on Chip (SoC) / low-power devices, which are equipped with specialized hardware, e.g., FPGAs or GPUs, that makes them suitable as servers for specific applications, but whose processing capabilities available for control / management activities is limited.
C. *It should adapt well to fast changing conditions:* in many use cases of practical interest the user devices are mobile and the application patterns are not known *a priori*, thus it is not possible to optimize once and for all the allocation of clients to edge nodes.
D. *It should be lean on backhaul resources:* the connectivity of the edge nodes, both between themselves and with core network components, called *backhaul* in telco terminology, may be scarce and heterogeneous.
E. *It should be cheap to maintain:* due to the sheer numbers of devices expected to be connected for future IoT applications, we argue that any sustainable business model must severely limit the expenses for operating and monitoring a deployed infrastructure, which in most cases will grow over time and remain in place for much longer than, e.g., mobile wireless access infrastructures, which have to catch up every few years with constantly advancing technologies.

3.2. Proposed architecture

Let us consider first **centralized solutions**, which are the baseline approach in cloud-based serverless solutions, such as Knative https://knative.dev or Apache OpenWhisk https://openwhisk.apache.org, and telco-native architectures, including the ETSI MEC as illustrated in Section 2.2. With this paradigm, illustrated in the top part of Fig. 3, the applications on the user devices merely obey to a logically centralized orchestrator, which instructs them to which end-point or URL to address their lambda requests. Since the entire decision making is done by the orchestrator alone, requirements A and B above are automatically covered. Also, since the orchestrator has a system view, we can expect that it can follow very well the changing conditions, possibly even anticipating such changes if prediction algorithms are used, which meets requirement C. For the same reason, requirement E is also addressed: the orchestrator is the only complex component of the system that needs be monitored and maintained. However, centralized solutions fall short in covering requirement D: making appropriate decisions require that the orchestrator is updated by all edge nodes on the real-time status of its resources. This may be reasonable in cloud-based solutions, where all the executors are powerful and well connected and in close proximity to one another in a data center, but it certainly poses limits to the growth of the edge system as the backhaul gradually becomes a choke point.

For this reason, in the literature some **distributed solutions** have been proposed (see Section 2.1). As illustrated in the middle part of Fig. 3, the basic concept of these approaches is that an overlay exists between the user devices and the executors, made by edge nodes that take local decision in a distributed manner to optimize the execution of services, which greatly reduces the internal traffic thus meeting requirement D. The user devices remain unaware of the underlying complexity, hence requirement A is met, as well. Adaptation to changing conditions (requirement C) is addressed, as long as the distributed system can reach near-optimum working point despite the decision makers have a limited view of the system. However, distributed solutions cannot address adequately requirements B and E. On the one hand, taking informed decisions in a fully distributed manner requires that the edge nodes coordinate among themselves and dedicate part of their computational capabilities to the process of maintaining a synchronized state for this purpose; therefore requirement B may be difficult to achieve, especially with a high number of edge nodes in the system. On the other hand, the edge system operator would have to maintain a potentially large number of active components in the system, including monitoring, supervision, and software upgrades phases, also addressing heterogeneous hardware and software characteristics, which makes it challenging to achieve requirement E.

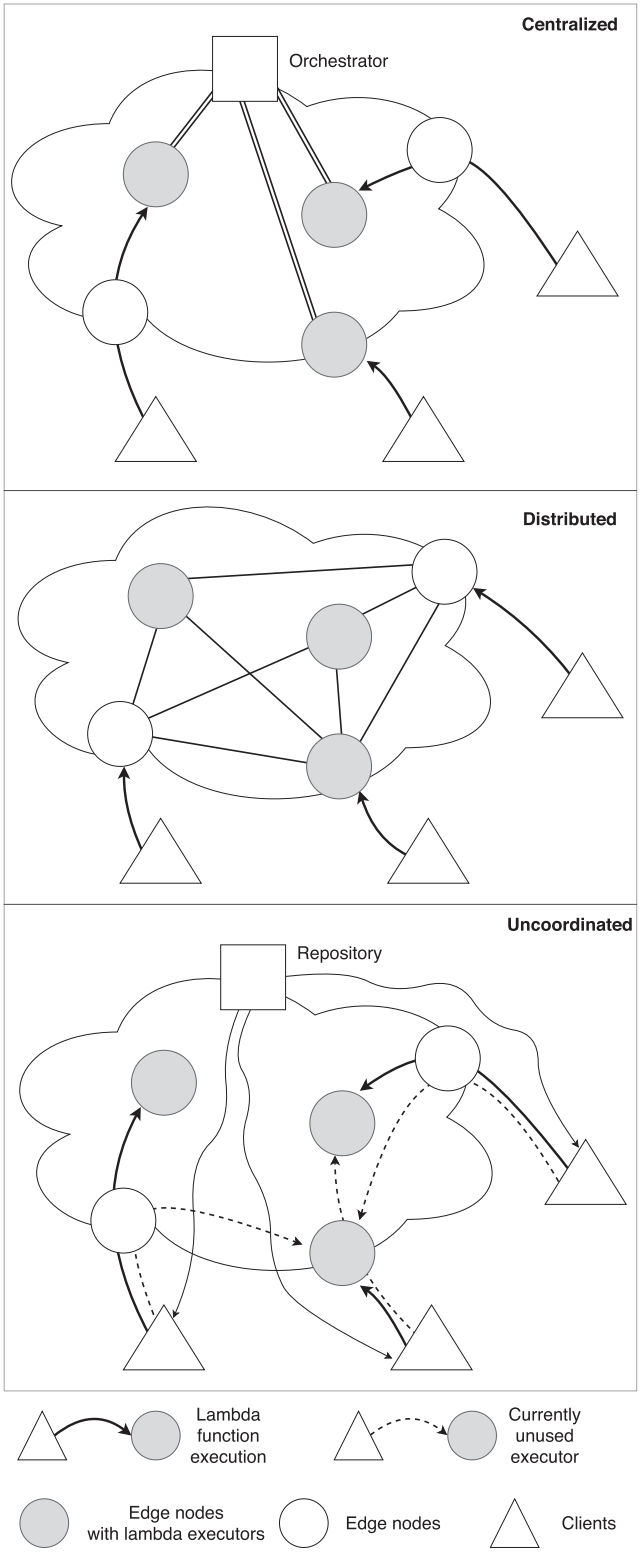

Fig. 3. Comparison of the proposed architecture for uncoordinated access to serverless functions (bottom) with traditional centralized (top) and distributed (middle) approaches.

Table 1
Qualitative comparison of the proposed architecture (uncoordinated) with classical centralized and distributed approaches from the literature, in terms of meeting five fundamental requirements (see Section 3.1).

Requirement	Centralized	Distributed	Uncoordinated
A. Easy to implement on the user side	++	++	+
B. Lean on edge compute resources	++	−	++
C. Adaptation to changing conditions	++	+	+
D. Lean on backhaul resources	−−	+	++
E. Cheap to maintain	++	−−	++

In this work we propose a solution, called here **uncoordinated access**, that overcomes the respective limitations of centralized / distributed approaches and addresses all the requirements. We start with an observation: slow-changing conditions in the system are easily detectable by a centralized entity, i.e., the orchestrator, with a low overhead since this merely requires aggregate measures from the executors and it is not a real-time task. Thus, let us assume that the lifecycle of the micro-service images on the executors is somehow optimized so as to follow *macroscopic* slow trends in the system. The real challenge is following the *microscopic* fast changes: if, for instance, an executor is installed in a SoC device, such as a Raspberry Pi, then very few concurrent executions of a lambda function can easily overload the executor, thus increasing the response times of clients and possibly degrading the application. Following these variations in a system with no reservation of resources nor *a priori* knowledge on the arrival of lambda execution jobs is extremely challenging, and in fact leads to their respective key shortcomings of the centralized and distribution solutions discussed above. On the other hand, rather than complicating the system to *beat* this variability, we propose to *embrace* it: we propose that the orchestrator allocates a pool of end-points / URLs to every client, which the latter can use to exploit opportunistically to its own advantage taking internal decisions. This is illustrated in the bottom part of Fig. 3, where solid lines represent the current choice of destination of clients and dashed lines are the (currently) unused alternatives they have been informed about by the central entity; the latter is called here *repository* to stress that it merely communicates a pool of executors to every client without running a real-time optimization process, as in a centralized architecture.

We call this solution *uncoordinated* because the clients do not interact with other components to take decisions on a per request basis. In an environment that evolves with fast dynamics, relying on the statistical multiplexing of uncoordinated agents taking myopic "good enough" decisions may result beneficial compared to a system trying to achieve "optimal" goals, which however fails because it is either fed outdated information or it consumes too many resources (computation, traffic) in the process. On the other hand, system-wide optimization can be added on top of the proposed solution, working at a much slower time scale (in the order of minutes and above). This can be done along at least the following two directions: modifying the set of executors deployed on edge nodes (e.g., an algorithm based on popularity of functions requested was proposed in [23]); advertising different pools of executors to the clients, based on long-term estimates of networking and computation statistics, which can be seen as a service placement problem (see [24]).

It is straightforward to see that the proposed uncoordinated solution meets all the requirements in Section 3.1, with the following two minor notes. First, this design only makes sense if the clients can decide which executor to use in a simple manner (requirement A), as we explain below in Section 3.3. Second, as for distributed solutions, we have to abandon the goal of achieving a global optimum, since this would require either a system-wide view or an extremely complex/expensive synchronization across the clients; however, we argue that "good" performance levels in a practical solution are way more preferable than reaching optimum performance under unfeasible conditions.

The above discussion is summarized for the readers' convenience in Table 1.

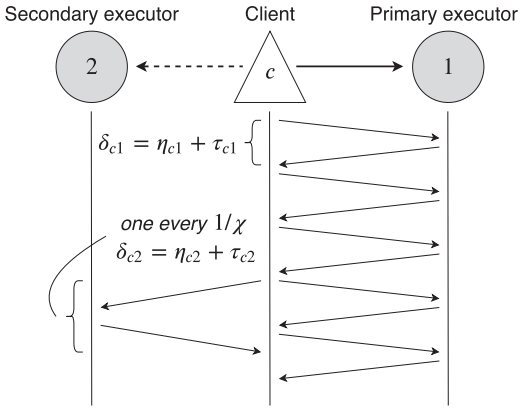

Fig. 4. Sequence diagram of the execution of lambda functions from a client towards its primary executor and, once every χ requests on average, to the secondary executor, as well, for probing purposes. In the figure δ is the overall delay, consisting of processing component η and network component τ; the full notation is introduced in Section 4 and summarized in Table 2.

3.3. Client algorithm

To complete our proposition we now describe how the clients select over time the executor to be used from the pool of those available. The pool of executors must be communicated to clients by a component with system-level view, indicated as a *repository* in Fig. 3, which in a real system would interact with the orchestrator in charge of managing the life cycle of executors on edge nodes. The algorithm that is used by (e.g.) the orchestrator to decide which pool of executors has to be notified to which client may be subject to optimization too, and is a research issue *per se*. However, since this happens at a time scale greater than that of interest for the scheduling of lambda functions, we consider this specific issue out of the scope of this work, and subject to future investigations. In the performance evaluation in Section 5, we select the pool of executors that minimize the number of network hops for a client to reach them, also limiting the pool size to 2 or 3.

To keep the client very simple, consistently with requirement A above, we propose that it keeps one of the possible destinations are the current one, then at every execution of the lambda function it selects also another destination for probing with a given probability χ, which is a system parameter. The lambda function is then issued both to the *primary* executor and to the *secondary* one selected: after measuring the relative latencies, the client may then promote the secondary to primary. An example sequence diagram is shown in Fig. 4. The primary executor (1) is depicted on the right of the client c, which issues lambda function requests to 1 only with probability $1 - \chi$. Sporadically, i.e., on average every $1/\chi$, one request will be issued towards both the primary executor (1) and the secondary executor (2), depicted on the left of c. In the example, the overall response time from 2 is greater than than from 1, hence the process continues at the client side as before.

In fact, the main reason for a network or service operator to use edge computing is because the applications have latency constraints. If this is

Table 2
Notation used.

Symbol	Domain	Meaning
C	$\{1, \dots, C\}$	Set of clients, $\lvert C\rvert = C$
$\mathcal{E}$	$\{1, \dots, E\}$	Set of executors, $\lvert\mathcal{E}\rvert = E$
S	$\{1, \dots, 2^C\}$	Set of states, $\lvert S\rvert = 2^C$
$\boldsymbol{\tau}$	$\mathbb{R}^{C\times E}$	Network delays
$\mathbf{x}$	$\mathbb{R}^{C}$	Task processor utilization
$\boldsymbol{\lambda}$	$\mathbb{R}^{C}$	Task arrival rate
$\boldsymbol{\mu}$	$\mathbb{R}^{E}$	Task dispatch rate
$\mathbf{s}$	$\{1, \dots, E\}^{C\times S}$	Primary executor per client per state
$\bar{\mathbf{s}}$	$\{1, \dots, E\}^{C\times S}$	Secondary executor per client per state
I_{ijk}	$\langle i, j, k\rangle \rightarrow \{0, 1\}$	Primary indicator function
$\bar{I}_{ijk}$	$\langle i, j, k\rangle \rightarrow \{0, 1\}$	Secondary indicator function
$\boldsymbol{\delta}$	$\mathbb{R}^{C\times S}$	Average delay with primary executor per client per state
$\bar{\boldsymbol{\delta}}$	$\mathbb{R}^{C\times S}$	Average delay with secondary executor per client per state
χ	$(0, 1)$	Probability that the secondary executor is probed
$\mathcal{J}_k$	$\subseteq S$	Set of states reachable by state k
P	$[0, 1]^{S\times S}$	Transition matrix of the associated DTMC
$\boldsymbol{\pi}$	$[0, 1]^{S}$	Stationary distribution

not the case, then cloud computing is bound to be a cheaper and easier alternative for economy of scale reasons. We note that the latency of a lambda transaction consists of several components, including the time to transmit the messages and the responses, network queuing delays between any two hops, and the lambda execution time plus any additional waiting due to the application/OS scheduler. However, from the point of view of the application on the user device, such decomposition is irrelevant: what counts is only the time between when the lambda function is issued and when a response is received, which can be easily measured locally.

Finally, the reader may wonder at this point why the lambda function is executed towards both destination instead of only the one under probing: this is to make sure in the most simple manner that the latency measurements are comparable. In fact, not all tasks of the same lambda type may be the same, e.g., the input may have a different size or contain data that are more or less complex to process on the edge node side; furthermore, environmental conditions may change from one lambda execution to the next one, e.g., the access link of the client may suffer from wireless impairments temporarily reducing the bit-rate.

3.4. ETSI MEC implementation

We now describe how to implement the proposed uncoordinated serverless scheme with ETSI MEC The reader is referred to Section 2.2 for a tutorial introduction to this standard.

As mentioned already, with serverless computing the executors do not hold a state for every active client application. This property can be exploited by the MEO to load the lambda images, e.g., Virtual Machine (VM) or containers, on the MEC hosts, according to slow-changing estimations / predictions of their utilization, which is outside the scope of this work. This way, all the interfaces from `Mm3` to `Mm7` (see Fig. 2 above) are not used either for execution of lambda transactions or for creation of new application contexts from device apps. The management plane, as such, is greatly simplified compared to traditional (stateful) applications, and in fact the MEO merely acts a repository of the end-points of the available lambda images on all the MEC hosts, grouped per lambda function type. Simplification also translates into a better scalability as the rate of context creation increases, which is a very desirable property in IoT scenarios where we can expect that some applications will have a short-lived duration.

As an application context creation from the device app is requested on the `Mx2` interface, via the LCM proxy, the MEO selects a number of executors and includes their end-points in the response to the device app.

The algorithm by which the MEO determines both the χ and which executors are to be selected for every new context is beyond the goal of this paper and part of our on-going research activities, fostered by the model illustrated in Section 4 below as a building block for the design of such an optimized algorithm in a production system.

4. System model and analysis

In this section we present a mathematical model of the uncoordinated serverless access system put forward in Section 3, under simplifying assumptions to make it tractable (Section 4.1). To facilitate the reader visualizing the model, we then study the simple case of 2 clients served by 3 executors (Section 4.2). Finally, we provide numerical results to compare a static allocation to our proposed solution and derive some system properties (Section 4.3). The conclusions found will be validated against experimental results in the next section Section 5.

4.1. System model

Despite the simplicity of implementation in both the client and the orchestrator, the proposed system is still too complex to be formalized in mathematical terms in general conditions. Therefore, we now make the following simplifying assumptions. We assume to have a set of C clients, each issuing lambda function requests of the same type towards a pool $\mathcal{E}$ of executors. For simplicity, we assume that both the arrival rate of tasks at every client and the serving rate at the executors are Poisson distributed, with average λ_i for client i and μ_j for executor j. We assume that the network delay τ_{ij} between any client i and executor j is constant and independent of the state of the system. Finally, we assume that the orchestrator provides every client with exactly two possible choices, which we call primary and secondary depending on which one is currently selected.

For consistency and better readability we adopt the following rules in the notation: the indices i and h always refer to clients, the index j to executors, the index k to states; vectors and matrices are indicated in **bold** (e.g., $\mathbf{x}$) and their corresponding elements use the same letter in regular font (e.g., x_i). A summary of the notation used is reported in Table 2.

Because of our assumptions above, we can consider the client and executors as a set of Markov M/M/1 queue systems. Without loss of generality we assume that the executors follow a Processor Sharing (PS) policy, which we believe to best approximate how a real edge node behaves under typical working conditions. Thus, for each client i we also define its processor share x_i. The equivalent M/M/1 system is illustrated by means of the example in Fig. 5 in the specific case where client 1 has executor 1 as primary and 2 as secondary, while the opposite applies to client 2. We call the set of these conditions a *state*, because it captures one of the possible combinations in which our system can be. If, for

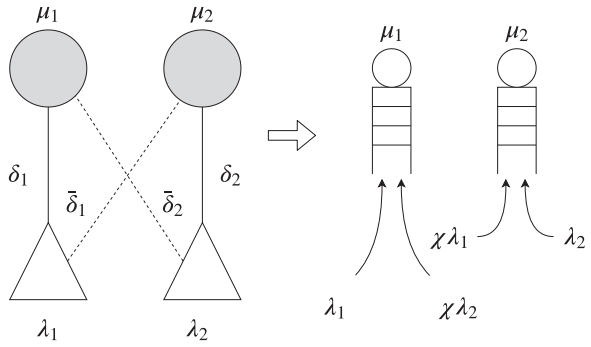

Fig. 5. System model, illustrated by means of an example with two clients and two executors, with $\boldsymbol{\tau} = \mathbf{x} = \mathbf{0}$; the equivalent M/M/1 model is on the right.

instance, the executor 2 becomes primary for client 1, then the system will be in a different state. The set of the possible states S is given by the binary enumerations of clients, because there are only two possible options (primary vs. secondary), thus $|S| = 2^C$. In any specific state k, such as that depicted in Fig. 5, we can identify the average delay δ_{ik} of any client i towards its primary executor, as well as the same quantity towards its secondary executor, called $\bar{\delta}_{ik}$. It is important to note that for every executor the inbound tasks are both those generated by the tasks that have it as the *primary* destination and those generated for probing reasons by the tasks that have it as the *secondary* destination. The latter follow the same Poisson distribution, but are only a fraction χ of the former.

According to basic queuing theory (e.g., [25]) the average delay in an M/M/1 PS system, including both queuing and processing times, is:

$$\eta = \frac{x_i \mu}{\mu - \sum_h \lambda_h}, \tag{1}$$

where λ_h is the arrival rate of any client h served by the executor, which only holds if $\mu > \Sigma_h \lambda_h$, i.e., if the system is *stable*. In our model we must take into account that, in any state s, an executor serves only the clients having it as a primary or secondary destination in that state. To capture this property, we define the following two *indicator functions*. First, I_{ijk} is 1 only if client i has primary executor j in state k, i.e., $s_{ik} = j$, otherwise it is 0. Likewise, $\bar{I}_{ijk}$ is 1 only if $\bar{s}_{ij} = j$, otherwise it is 0. We can now express the average delay of client i towards its primary executor when the system is in state k as follows, based on Eq. (1):

$$\delta_{ik} = \frac{x_i \mu_{s_{ik}}}{\mu_{s_{ik}} - \lambda_i - \sum\limits_{h \in C, h \neq i} \lambda_h \left[I_{hs_{ik}k} + \chi \bar{I}_{hs_{ik}k} \right]} + \tau_{is_{ik}}, \tag{2}$$

and, similarly, the average delay of client i towards its secondary executor when the system is in state k:

$$\bar{\delta}_{ik} = \frac{x_i \mu_{\bar{s}_{ik}}}{\mu_{s_{ik}} - \chi\lambda_i - \sum\limits_{h \in C, h \neq i} \lambda_h \left[I_{h\bar{s}_{ik}k} + \chi \bar{I}_{h\bar{s}_{ik}k} \right]} + \tau_{i\bar{s}_{ik}}, \tag{3}$$

Both Eqs. (2) and (3) assume that the queues are stable, i.e., that the respective denominator is positive. If this condition is not true, then the queue length grows over time and the average delay tends to infinity in theory, while in practice this condition will lead to much higher delays than usual.

Right up to this point we have shown how to build the two matrices $\boldsymbol{\delta}$ and $\bar{\boldsymbol{\delta}}$, which give us the average delays experimented by every client towards its two possible destinations. We now use this information to infer the average behavior of the system at a steady state and, hence, derive the average delay of every client. Let us consider that the real system is dynamic: every client randomly performs probing on the primary vs. secondary executor, based on which it decides whether it should swap their role. If we assume that every client takes decisions based on the *average* delay, as expressed in Eqs. (2) and (3), we see that the next state *for a given client* i is fully determined: if $\delta_{ik} > \bar{\delta}_{ik}$, then i will continue using s_{ik} as its primary executor; otherwise, the new primary executor will become $\bar{s}_{ik}$. However, this is an uncoordinated systems where all the clients take their decisions individually, thus the transition from any state k_1 to k_2 is determined by the random times when all the clients take their swap decisions. In other words, it is a stochastic process, which we can represent by means of an associated Discrete-Time Markov Chain (DTMC), where each state is exactly one of the possible states S of our system, and the transition matrix P is built as follows. For every state $k \in S$ we consider all possible states J_k that can be reached, where state $z \in J_k$ iff the system can go from k to z with a combination of clients i changing their primary executor because of $\delta_{ik} > \bar{\delta}_{ik}$:

$$J_k = \left\{ \forall z \in S, z \neq k | \forall i : \left(\delta_{ik} \leq \bar{\delta}_{ik} \wedge s_{ik} = s_{iz} \right) \right\} \tag{4}$$

To simplify notation, we assume that all the queues are stable, in both the primary and the secondary executors.[2] If there is a state k such that all the delays to the primary executor are smaller than the delays to the secondary executor (i.e., if $\exists k : \forall i, \delta_{ik} \leq \bar{\delta}_{ik}$), then it is $J_k = \emptyset$. In this case k is an *absorbing state*; in our system this means that every client sees the secondary destination as a worse option compared to the primary destination, thus it does not swap the two, and the system remains stable indefinitely. In general, the cardinality of this set is given by:

$$|J_k| = 2^{|\{\forall i, \delta_{ik} > \bar{\delta}_{ik}\}|} - 1 \tag{5}$$

It may also happen that state k is *unreachable*, i.e., $\forall z : z \notin J_k$; an unreachable state will not be reached unless the system starts from it.

Once all the J_k are determined for each $k \in S$, the transition probability p_{kz} from state k to state z in the matrix P is:

$$\forall k, z \in S, p_{kz} = \begin{cases} 0 & \text{if } z \notin J_k \\ \frac{1}{|J_k|} & \text{if } z \in J_k \end{cases} \tag{6}$$

Without considering the systems with absorbing states, which are of little practical interest to our analysis, and after removing the unreachable states, we obtain a chain that is irreducible (i.e., it is possible to go from any state to any other), and whose states are positive recurrent by construction. Thus, the chain has a positive unique stationary distribution $\boldsymbol{\pi}$, which gives the average probability in the long term that the system will be in any given state. Finally, we can use the latter to derive the average delays per client as $\boldsymbol{\delta\pi}$, i.e.:

$$E[\delta_i] = \sum_{k=1}^{S} \delta_{ik} \pi_k \tag{7}$$

To better visualize the process and system variables we report in the following a simple numeric example.

4.2. Example

We now report a small numeric example, with the only purpose of guiding the reader towards an easier understanding of the proposed notation and model. We have two clients (1 and 2) and three executors (1, 2, and 3), whose arrival and serving rates, and network delays, are shown in Fig. 6 and given below:

$$\begin{aligned} \boldsymbol{\lambda} &= \begin{bmatrix} 3 & 4.5 \end{bmatrix} \\ \boldsymbol{\mu} &= \begin{bmatrix} 5 & 10 & 15 \end{bmatrix} \\ \mathbf{x} &= \begin{bmatrix} 1 & 1 \end{bmatrix} \\ \boldsymbol{\tau} &= \begin{bmatrix} 1 & 1 & 2 \\ 1 & 1 & 2 \end{bmatrix}. \end{aligned} \tag{8}$$

[2] In the numeric analysis in Section 4.3 below, we have taken into account unstable queues as follows: $\delta_{ik} > \bar{\delta}_{ik}$ only if δ_{ik} is finite (i.e., stable queue towards the primary executor), in which case the condition is always true if $\bar{\delta}_{ik}$ is infinite (i.e., unstable queue towards the secondary executor).

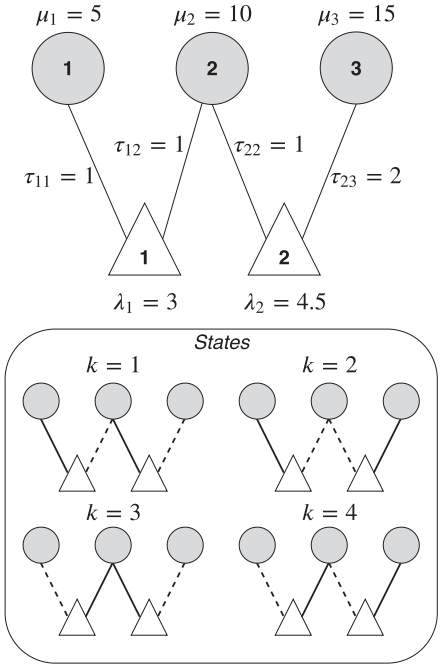

Fig. 6. Toy example of a system with two clients and three servers illustrated in Section 4.2 to visualize the data structures involved in the model.

As can be seen: client 2 has a heavier load but it can use faster executors; the fastest executors, i.e., 3, has the highest network delay.

First, we enumerate all possible 4 states, which are illustrated graphically in the bottom part of Fig. 6 and formally determined in our notation as:

$$\mathbf{s} = \begin{bmatrix} 1 & 1 & 2 & 2 \\ 2 & 3 & 2 & 3 \end{bmatrix}$$
$$\bar{\mathbf{s}} = \begin{bmatrix} 2 & 2 & 1 & 1 \\ 3 & 2 & 3 & 2 \end{bmatrix}. \tag{9}$$

So far, we have simply defined the structures holding the *input* of our problem. Let us now move to the analysis, starting with determining the average delays in the current vs. probing destination for all clients in all states, using Eqs. (2) and (3), respectively:

$$\boldsymbol{\delta} = \begin{bmatrix} 3.5 & 3.50 & 5.00 & 2.52 \\ 2.9 & 3.42 & 5.00 & 3.42 \end{bmatrix}$$
$$\bar{\boldsymbol{\delta}} = \begin{bmatrix} 2.92 & 2.08 & 2.06 & 2.06 \\ 3.03 & 2.08 & 3.03 & 2.52 \end{bmatrix}. \tag{10}$$

Based on the $\boldsymbol{\delta}$ and $\bar{\boldsymbol{\delta}}$ average delays, we can then write down all the $\mathcal{J}_k$ sets of states that can be reached from state k. For instance, $\mathcal{J}_1 = \{3\}$ because, by looking only to the first column of $\boldsymbol{\delta}$ and $\bar{\boldsymbol{\delta}}$, we see that for client 1 in state 1 it is better to move to its secondary executor because $\delta_{11} > \bar{\delta}_{11}$, whereas for the client 2 in state 1 it is better to stick to its primary executor since it is $\delta_{21} < \bar{\delta}_{21}$; thus, the only possible transition is from state 1 to state 3, see also the graphical representation of the states in Fig. 6. Based on the formal definition of $\mathcal{J}_k$ in Eqs. (4) and (6), we can build the transition probability P as:

$$P = \begin{bmatrix} -1 & 0 & 1 & 0 \\ 0.33 & -1 & 0.33 & 0.33 \\ 0.33 & 0.33 & -1 & 0.33 \\ 0.33 & 0.33 & 0.33 & -1 \end{bmatrix}, \tag{11}$$

which is irreducible and with positive recurrent states, and it has the following stationary distribution:

$$\boldsymbol{\pi} = \begin{bmatrix} 0.25 & 0.19 & 0.37 & 0.19 \end{bmatrix}. \tag{12}$$

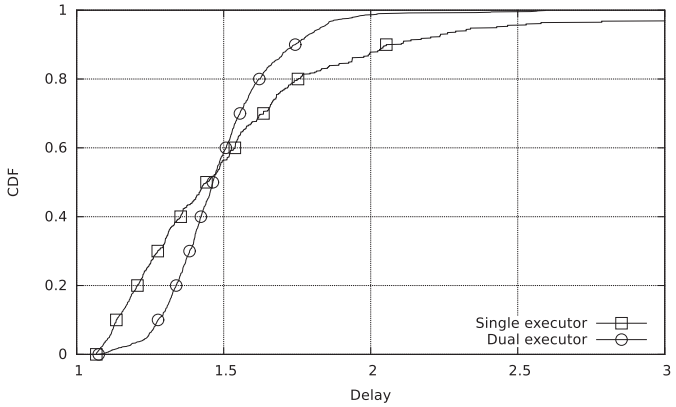

Fig. 7. Delay distribution in single vs. dual executor scenarios, with 10 clients and 6 executors.

Eventually, the average delay per client are determined by multiplying $\boldsymbol{\delta}$ by $\boldsymbol{\pi}$ which gives:

$$E[\delta] = \begin{bmatrix} 3.88 & 3.89 \end{bmatrix}, \tag{13}$$

therefore in this example the two clients have a very similar average delay.

4.3. Analysis

In this section we report a numeric analysis obtained with the model described in Section 4.1 above. The tools and scripts used are available as *open source* software on GitHub https://github.com/ccicconetti/markovsim.

In a first batch of results, we compare our uncoordinated serverless access scheme to the baseline solution of statically allocating clients to executors. In both cases, the association between the client and its only executor (or its primary and secondary executors) is random. We measure the performance in terms of the average delay of clients, which is given by Eq. (1) for the single executor case and by Eq. (7) with dual executors. We evaluate the performance as the load grows, by increasing the number of clients from 2 to 14; on the other hand, we consider 4, 6, and 8 executors, respectively, while keeping the overall serving rate equal to 96 (i.e., with 4 executors each has a $96/4 = 24$ serving rate, etc.). For simplicity, we consider that the network delay is negligible for every client-executor pair. The value of χ is always 0.1. For each combination of parameters we ran 100 independent runs. In each run we draw randomly the load of every client from 1 to 3, each with the same request duration equal to 1.

In Fig. 7 we plot the Cumulative Distribution Function (CDF) of the delay, in a random but representative combination of 10 clients and 6 executors. We can see that while the median in the two cases is almost the same, the dual executors distribution is much less skewed than that with a single executor: the probing mechanism in the uncoordinated serverless access proposed is very effective in keeping the delay of clients within a smaller range, even with a random assignment of clients to executors, which is clearly a worst case. This property is especially important for those IoT applications that rely on the response time for the execution of a remote function being upper bounded for correct/smooth operation.

In Fig. 8 we summarize the results obtained is all the combinations studied, by reporting only the 95th percentile of the average delay of the clients. First of all we note that the curves decreases as the number of executors decreases, for both the single executor and the dual executors: since the *overall* serving rate is the same, it is expected that having a smaller number of executors reduces the probability that a single executor becomes overloaded as a result of an uneven allocation of clients to executors. Second, as can be seen, the single executor curve always

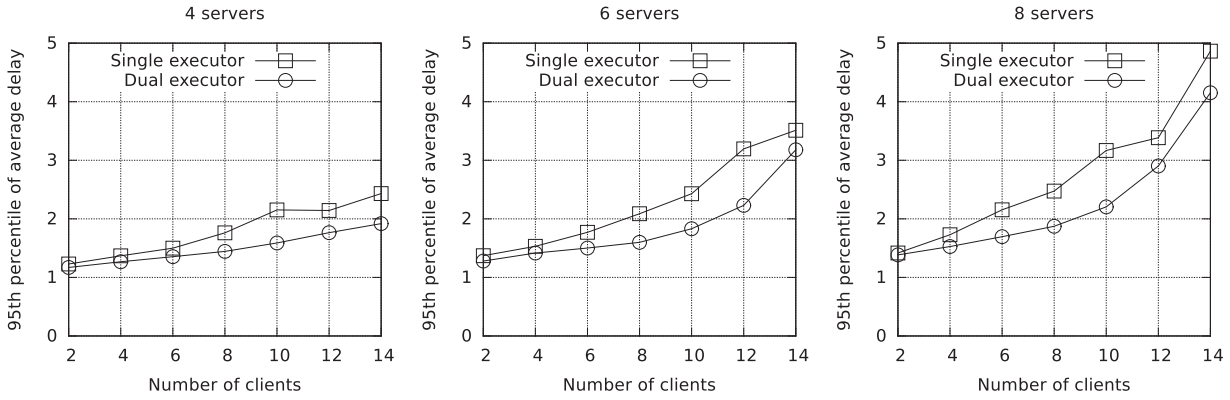

Fig. 8. Comparison between single vs. dual executor with 4, 6, and 8 executors and an increasing number of clients, in terms of the 95th percentile of the average delay of the users.

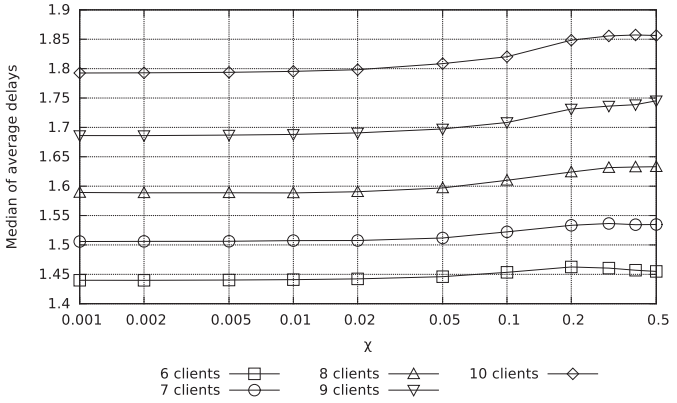

Fig. 9. Median of average delays with increasing χ with different number of clients.

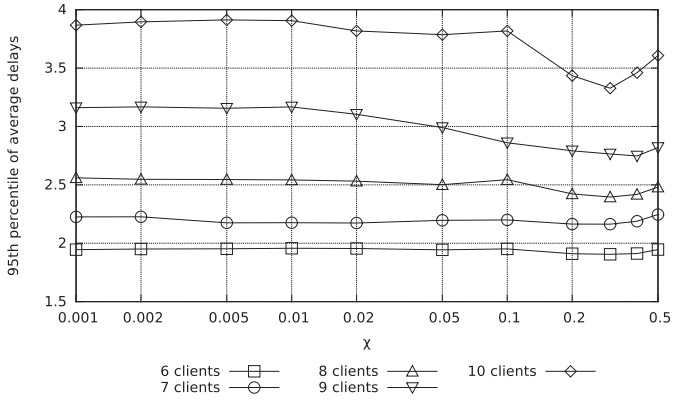

Fig. 10. 95th percentile of average delays with increasing χ with different number of clients.

lies on top of that with dual executors, which confirms the behavior described for the specific case in Fig. 7 in all the combinations tested.

We now study another scenario, with the goal of assessing the impact of χ on the system dynamics. We keep the number of servers constant and equal to 6. Also, the client load is equal to 2, whereas the serving rate of the executors is drawn uniformly between 8 and 16. We increase the number of clients from 6 to 10, and the value of χ from 0.001 to 0.5. In this case we ran 1000 independent replications for every combination of the factors.

First, in Fig. 9, we show the median of the average delays of all clients with increasing χ and number of clients. As can be seen, the delay is not very sensitive to large changes of χ in the range under test, which is positive because we can expect this parameter not to have a crucial relevance in the overall system configuration. However, especially at higher loads, we can see that high values of χ tend to exhibit a higher median average delay.

We then show the 95th percentile of the average delays in Fig. 10. Like the median, the 95th percentile is not affected significantly by changes of χ below 0.01. However, with higher values, and again especially at higher loads, the 95th percentile of the average delays decreases as χ increases. Intuitively, the reason for this is that at high loads *exploration* becomes more important because there is a high chance that, due to uneven allocation, one of the executors is heavily loaded. In other words, when increasing χ the overall system load also increases, because the clients do more probing, but, depending on the conditions, the extra load can be useful as it benefits users that would otherwise spend too much of their time with an overloaded primary executor. As can be expected, there is a trade-off: as the value of χ becomes too high, then the delay increases again because overall the system becomes overloaded.

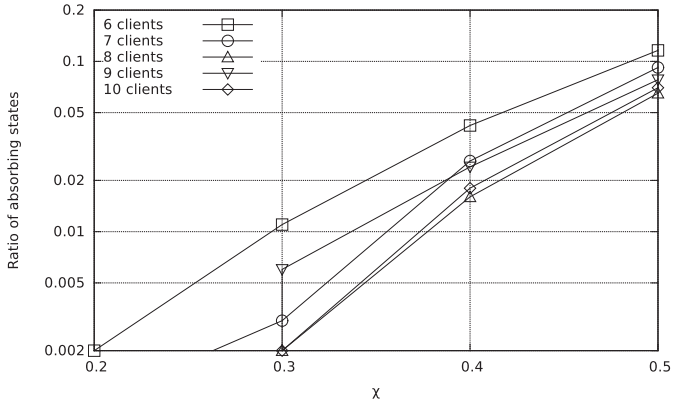

Fig. 11. Ratio of experiments with absorbing states.

Another effect of the value of χ being too high is that many states of the system have unstable queues, thus leading to a much sparser transition matrix P in our model. This, in turn, fosters the appearance of absorbing states, which are otherwise extremely rare, as can be seen in Fig. 11, which shows the ratio of scenarios leading to a transition matrix with an absorbing state over the total number of replications in the same conditions. With $\chi < 0.2$ the ratio is 0, hence it is not shown in the graph, but it increases steeply (note the y-axis logarithmic scale in this plot) after $\chi = 0.2$, for all the number of clients. We leave as future work deeper elaborations on predicting the conditions leading to a transition matrix with an absorbing state and on its system performance impact.

5. Performance evaluation

In this section we study the performance of the proposed solution for uncoordinated serverless access with a testbed implementation in an emulated edge network. We first introduce the methodology and tools used for the evaluation (Section 5.1). Then we discuss the results obtained in two scenarios aimed at different objectives. In Section 5.2 we validate the qualitative conclusions from the model analysis in Section 4.3 in a non-realistic scenario that mimics the system model defined therein. In Section 5.3 we set up an environment in realistic conditions to assess the performance of our uncoordinated serverless access scheme, compared to alternative state-of-the-art solutions.

5.1. Methodology and tools

In this paper we re-use the performance evaluation framework described in [18], briefly summarized in the following. Performance evaluation of edge systems is a challenging task: full-scale deployments are most accurate but they require a huge effort for the realization and may seldom be configured in such a way to run fully repeatable experiments; cloud simulators are very versatile but they focus on modeling adequately only one or few aspects of the system (e.g., data placement in [26] or scheduling in [27]); finally, packet-level simulators (e.g., [28]) include realistic models for the communication but cannot easily accommodate real applications. We believe that our approach achieves a good trade-off between accuracy of results under realistic condition and execution in a controlled and repeatable environment, by using real applications running in lightweight containers emulating a real network with mininethttp://mininet.org/. The clients and servers are written in C++ and they communicate via REST interfaces, realized with the popular gRPChttps://grpc.io/ library from Google.

For scalability reasons, lambda executors do not perform computations based on the input, but instead simply emulate the behavior of an application running in a VM with given virtual resources assigned, in terms of number and speed of CPUs and amount of memory available, processing incoming requests with a pool of pre-allocated workers, where waiting tasks are served with a First Come First Serve (FCFS) policy. In both scenarios below we have configured the lambda executor emulators so that a single worker fully using its CPU requires 50 ms processing time for a lambda request of size 5,000 bytes. We have carried out a sensitivity analysis to verify that the conclusions are not affected by this particular choice, as well as by some others listed in the respective scenarios (including the lambda request rate and the number of executors). The results are however not reported in this work because they do not provide the reader with significant insights on the matters under study.

We have implemented the following solutions for comparison reasons:

- **uncoordinated-2/uncoordinated-3**: the uncoordinated serverless access proposed in Section 3, with two and three possible destinations, respectively; based on preliminary results (included in the supplementary material) additional destinations yield inferior performance in the scenarios we have considered; recall that in our mathematical model in Section 4 we limited ourselves to just two destinations to keep it tractable;
- **static**: the allocation of every client to just one executor, as the baseline approach in edge computing also implicitly assumed by the ETSI MEC;
- **centralized**: a single node in the network performs load balancing, using a weighted round-robin policy, where the weight is equal to a running estimate of the execution latency towards the given destination; this approach is illustrated at a high level in the example in the top part of Fig. 3;
- **probing**: same as centralized, but lambda dispatching happens by first querying all the executors on the processing time required if no other task is received, then selecting the one reporting the shortest duration; this approach is proposed in [29] as a solution for scheduling tasks in an edge-cloud system;
- **distributed**: same as centralized, but there are dispatchers distributed over the network that perform load balancing purely based on their local information to limit the communication overhead, as we proposed in [18].

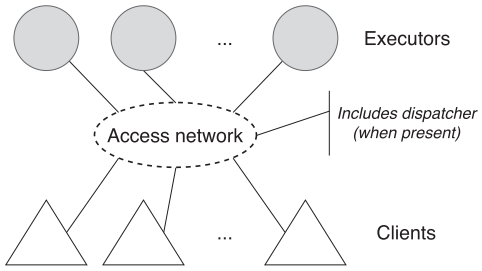

Fig. 12. Small-scale scenario: network topology.

Every experiment has been repeated with the same configuration, but different seeds for the initialization of random number generators, until achieving statistical convergence. In the plots we report 95% confidence intervals where appropriate for the type of experiment and unless they are negligible compared to the respective mean values. Each experiment lasts for 70 seconds or 310 seconds, respectively in Section 5.2 and Section 5.3, but the initial 10% is always considered as warm-up and the measurements in that period are discarded. The value of χ in all the experiments reported below is constant and equal to 0.1 as a compromise between reacting fast to changing conditions (which would require χ as big as possible) and keeping the probing overhead reasonable (overhead increases with χ). The value was found based on a preliminary analysis whose results are not reported in the paper but available as part of the supplementary material.

5.2. Small-scale scenario

In this section we aim at validating the conclusions inferred from the numerical analysis of mathematical model in Section 4.3: uncoordinated serverless access brings advantages, in terms of the high percentiles of delays, compared to static allocation of clients to executors, despite it increases the overall system load. Specifically, we set to achieve this goal in a topology that clearly benefits a centralized or distributed solution (defined in Section 3.2): as illustrated in Fig. 12, a single access network separates the clients from the edge nodes, also providing a perfect "natural" location for a load balancer. As a matter of fact, since both the centralized and distributed policies here would have a single load balancer, there is no distinction between them and, thus, they are identified as a single case in plots. All links have a 100 Mb/s capacity with 1 μs delay. The clients continuously issue an average of 5 lambda requests per second following a Poisson distribution. The number of servers is always 8 while the number of clients is increased from 16 to 32, which also increases proportionally the overall load. In every experiment we select randomly the number of CPU cores available per executor. For the static, uncoordinated-2, and uncoordinated-3 policies, the set of target destinations per executor is selected randomly in every experiment; for the others, the load balancing node is located in the access network, which is the more natural placement providing best results.

In Fig. 13 we show the cumulative distribution of the *delay*, which is defined as the time between when a client issues a lambda request towards the destination (or the load balancer, when present) and when it fully receives back the response. With uncoordinated-2 and uncoordinated-3 the multiple lambda requests are fired in parallel and the delay stops as the first response is received from the executor requiring the least processing + networking time, with further responses being

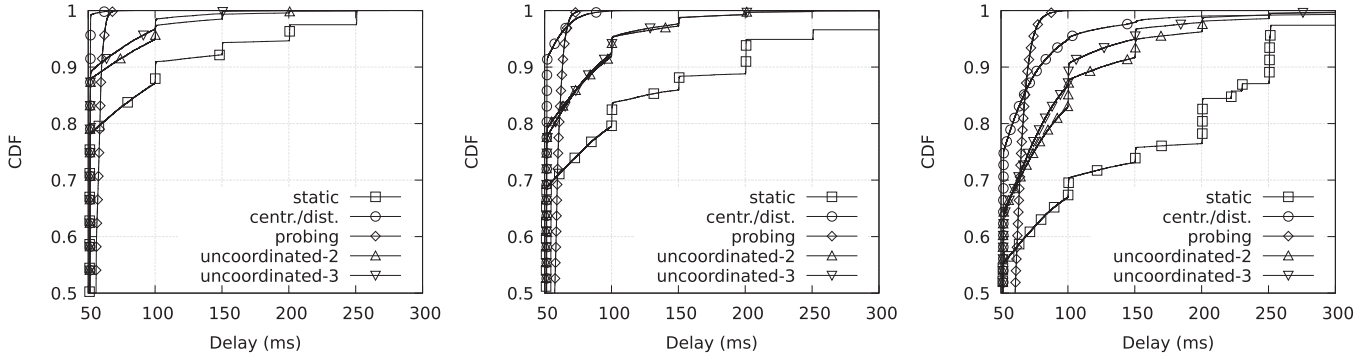

Fig. 13. Small-scale scenario: delay distribution with 16 (left), 24 (center), and 32 (right) clients.

simply discarded. Every curve has been obtained by putting together all the delays of all clients in all independent repetitions run with a given combination of number of clients and policy used. Thus, confidence intervals are not applicable to this metric.

As we can see from the plots, the probing policy achieves excellent results at all traffic loads. This is because the centralized entity, which in this topology is perfectly located, asks the executors about the processing time of every incoming lambda requests, which makes the mechanism robust to both uneven allocation of computation resources and temporary congestion due to unbalanced traffic. As a matter of fact, in [29] the authors prove that such a scheduler is $(1+\epsilon)$-speed $\mathcal{O}(1/\epsilon)$-competitive, which is extremely good in a system where it has been proved that no online algorithm can be optimal. Unfortunately, this solution cannot be implemented in practice because, in general, an executor does not know beforehand the time required for the execution of a function. Also, the traffic overhead caused by this approach is significant, as will be seen later. Thus, we consider probing merely as an ideal performance reference.

Load balancing, indicated in the plots as centr./dist., is instead a viable solution, which yields a smooth, but relatively small, increase of the delay as the load increases from 16 to 32 clients. The attentive reader may have noticed that centr./dist. achieves even *better* performance than probing with only 16 clients: this is because the former is not encumbered by having to ask the executors about the future processing time, as the latter is required to do. The performance of our proposed uncoordinated scheme is only marginally worse than that of centr./dist., which in our opinion is very remarkable because it does not require any additional architectural element that would add complexity (hence development and maintenance costs), hamper the scalability, and become a single point of failure, as elaborated extensively in Section 3. In this scenario the difference between uncoordinated-2 and uncoordinated-3 is only slight with 32 clients and negligible at lower loads. Finally, a static allocation exhibits poorest performance by far: as already evident from the results of the numerical analysis in Section 4.3 in simplified conditions, adding just one more destination option greatly improves the performance in terms of delay, especially at high percentiles, which are most important in latency-sensitive IoT applications.

In Fig. 14 we show the overall *network traffic*, defined as the sum of the average traffic in the unit of time of all the network links. In this work, the metric is an indirect measure of the overhead incurred by the various strategies adopted: since there are no other ongoing transmissions between nodes in our experiments, under the same rate of lambda requests served, if solution A has a higher network traffic than solution B it means that A required additional data exchanges compared to B. As introduced earlier, we see that probing has a huge network overhead, even in such a small-scale topology as that considered. On the other hand, static has the lowest traffic requirement, which is rather obvious since the clients transmit to a single executor (unlike uncoordinated-2 and -3) and without the need to maintain an overlay, as with a distributed approach. The uncoordinated solutions exhibit a slightly higher network

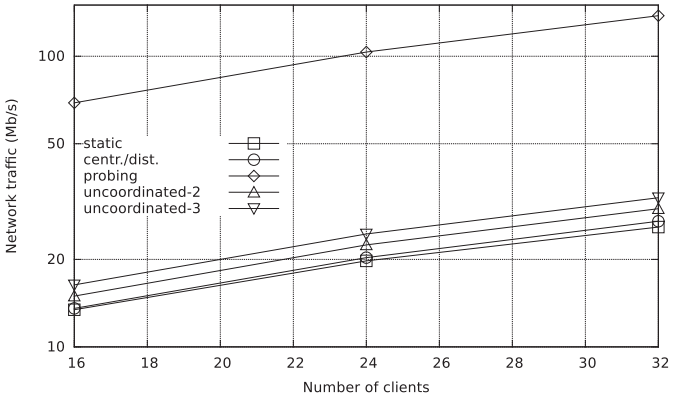

Fig. 14. Small-scale scenario: network traffic.

traffic, which creates the following trade-off: the higher the number of destinations (and the higher the value of χ), the lower are high quantiles of delay but the higher is the communication overhead. Depending on the target environment and expected Quality of Service (QoS) requirements of the applications, a suitable calibration must be done by the edge system operator to achieve best performance.

5.3. Large-scale scenario

In this section we use a large scale scenario in a topology extracted from a real IoT network: Array of Things https://arrayofthings.github.io/, a collaborative effort with about 100 nodes installed at intersections in Chicago, IL, US, using the Waggle platform [30], which is more realistic than both the system model in Section 4.1 and the environment in the previous experiments (Section 5.2). Starting from the geographical locations of the real nodes, we have first collapsed nodes that are too close to one another, then added bi-directional 100 Mb/s capacity / 1 µs latency links between nodes based on a threshold distance. The resulting network map is illustrated in Fig. 15 and it consists of 45 nodes with a diameter of 11 hops.

In this scenario the clients adopt the following traffic pattern: a burst of lambda requests is generated at the beginning of consecutive periods, whose duration is drawn from an exponential distribution with mean 10 s. The burst size, expressed in terms of number of lambda requests, is drawn from a Poisson distribution with mean 25. After receiving the response, the client backs off for a random amount of time, drawn from a uniform distribution in [150, 200] ms before issuing the next request, to model processing on the client side. Like in the previous scenario, the executor emulators and client applications are configured in such a way that a single task requires 50 ms to be executed on a given core with no other concurrent task being processed. Thus, the duty cycle at low loads is about 0.5 ($= 25 \times (50+150)/10^3$).

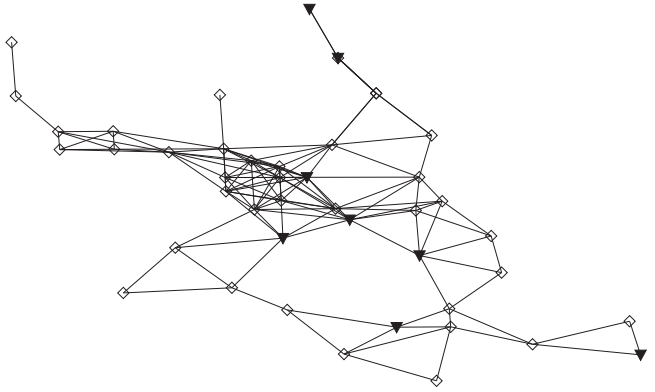

Fig. 15. Large-scale scenario: sample network topology showing servers (circles) and clients (triangles), both also acting as intermediate networking devices.

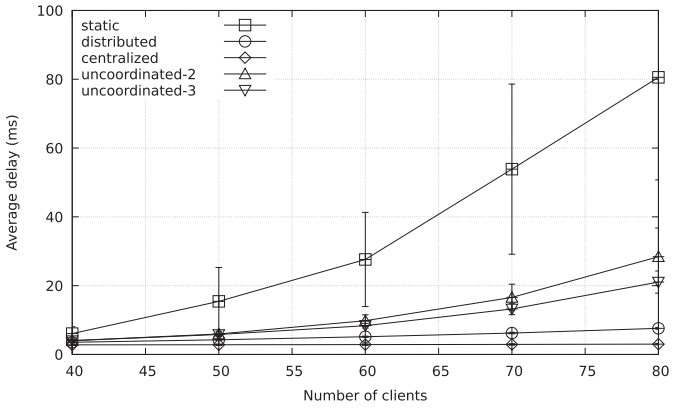

Fig. 16. Large-scale scenario: average delay.

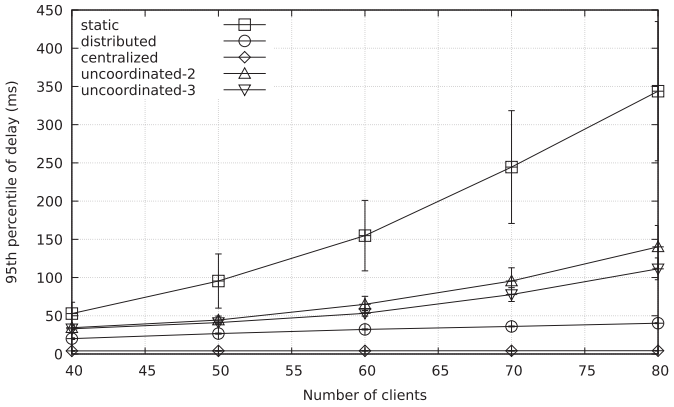

Fig. 17. Large-scale scenario: 95th percentile of delay.

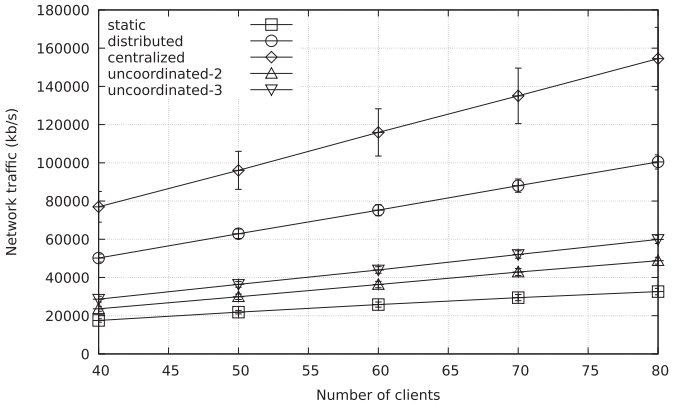

Fig. 18. Large-scale scenario: network traffic.

In every replication, 8 out of the 45 nodes are selected as executors, each with two CPU cores and workers. All the nodes may host clients, which are dropped randomly at the beginning of every experiment. The number of clients is increased from 40 to 80. In uncoordinated-2 and -3, the destinations are selected among those having a shortest path from the client; for instance, with uncoordinated-2 if there is destination A three hops away and destinations B, C, and D four hops away, then the rest are farther, we select randomly two destinations out of {A, B, C, D}. By extension, with static we always select the closest executor, breaking ties randomly when required. With centralized we select randomly the node acting as load balancer and all the clients contact the executors through the latter. With distributed we assume that all the nodes hosting an executor also host a distributed dispatcher; clients always contact the closest dispatcher for the execution of lambda requests. In this scenario, when using a probing policy the system becomes unstable, i.e., the traffic consumed by the central load balancer for polling all the executors to determine which one is best suited to serve the next incoming lambda request is so high that the communication links are saturated, which leads to ever-growing queues (and delays) of client requests. This confirms the impossibility to implement the probing policy in a practical scenario. Results with probing are not shown in plots.

As in the previous section, the key performance index for this scenario is the delay, which in this section is subtracted a constant value equal to the minimum processing time of the lambda requests, i.e., 50 ms, for better readability of plots. As can be seen in Fig. 16, uncoordinated-2 and -3 achieve intermediate performance in terms of the average delay, rather close to that of distributed and centralized, while the static curve lies well above the rest. This behavior is exacerbated for the 95th percentile of the delay, reported in Fig. 17. This confirms that also in a more realistic topology with bursty traffic an uncoordinated serverless access provides a significant advantage, in terms of delay, compared to a static allocation of clients to executors. Performance can be improved further by using more sophisticated policies, which however require new components and have a higher network overhead.

The last statement is proved in Fig. 18, which shows the overall network traffic. Unlike the previous scenario, which was very optimistic for the centralized/distributed policies, the network overhead of both is significantly higher than that of uncoordinated access. This is because, without a natural central node in the network, the use of an overlay for dispatching lambda tasks can be very expensive. Instead, the price to be paid by uncoordinated-2 and -3 compared to static, in terms of network traffic, is limited, and deemed to be affordable in most cases because of the advantages it brings in terms of delay and reliability.

We conclude the analysis with the average *utilization* of the executors, defined as the ratio between the time an executor is busy processing at least one task and the experiment duration, in Fig. 19 (confidence intervals here are omitted because negligible). This is to show that uncoordinated access requires an additional price: computational resources on the executors must be invested to process lambda requests without a strict necessity to do so. In fact, while distributed, centralized and static have almost overlapping performance, the utilization becomes higher with uncoordinated-2 and even more so with uncoordinated-3. We leave for future studies the design of more sophisticated policies that retain most advantages of the uncoordinated access techniques, while also reducing their network and computational overhead.

6. Conclusions and future work

In this paper we have investigated the problem of fast changing connectivity and computational load conditions for serverless IoT applications running in an edge network. Based on a critical analysis of

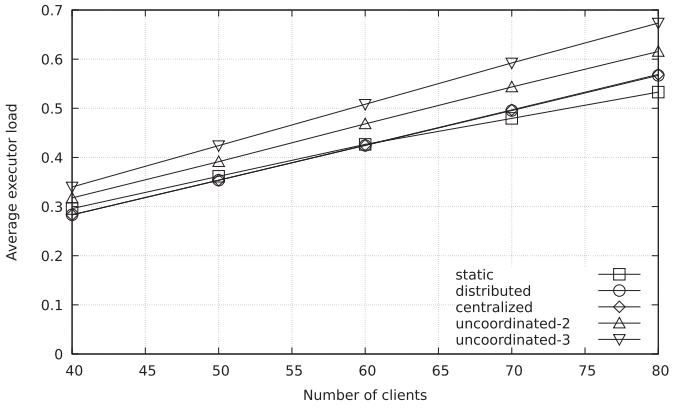

Fig. 19. Large-scale scenario: average executor load.

state-of-the-art solutions relying on either centralized dispatching or a distributed overlay, we have proposed a new approach called *uncoordinated serverless access*, which does not require complex/costly/fragile system elements, hence it is practical to implement in fast growing and fragmented IoT deployments. We have developed a mathematical model using queuing theory under simplified assumptions, which shows that the proposed approach reduces the delay of response times compared to a static allocation of clients to micro-service executors, which is today's baseline. These numerical results have been confirmed by experiments carried out with a prototype implementation in two emulated networks, one of which uses a realistic topology and bursty traffic. The goal achieved is especially important for latency-sensitive applications, which can be found in many areas of huge practical interest, such as connected car and industry automation. In the emulation experiments we have also compared uncoordinated serverless access with centralized load balancing and distributed dispatching: the results have shown that our proposed solution, in addition to being simpler and requiring fewer maintenance, requires much less network traffic (−65% than centralized, −55% than distributed) while requiring only +10% computational load on executors. Finally, the uncoordinated access scheme proposed can be realized within the ETSI MEC.

Even though distributing computing has been extensively studied in the scientific literature and is a mainstream technology for cloud systems, very few of the models and technologies apply to IoT systems, especially when used in edge networks, which are emerging as the most viable approach to a sustainable deployment in several business areas. In our opinion, what we have presented in this paper is only the beginning in a new area of research on how to design, operate, optimize, and maintain complex systems where IoT devices consume services offered by heterogeneous devices close to them with limited compute and connectivity capabilities.

With specific reference to this work, we believe it could be extended in at least the following directions: further elaboration on the mathematical model to infer actionable properties in some specific conditions (e.g., with homogeneous population of clients); closed-loop systems to modify at run-time the system parameter configuration (e.g., χ or the number of destinations per client); more sophisticated stateful algorithms (e.g., including prediction/estimation) to be used by clients that are powerful enough; integration with orchestration systems for an optimized selection of the destinations of each client beyond shortest-path.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.comnet.2020.107184.

CRediT authorship contribution statement

Claudio Cicconetti: Software, Formal analysis, Writing - original draft, Visualization. **Marco Conti:** Supervision, Writing - review & editing. **Andrea Passarella:** Methodology, Validation, Writing - review & editing.

References

[1] IGR, The Business Case for MEC in Retail: a TCO Analysis and its Implications in the 5G Era, Technical Report, 2017.
[2] 5GAA, Toward Fully Connected Vehicles : Edge Computing White Paper, Technical Report, 2017.
[3] 5G-ACIA, 5G Non-Public Networks for Industrial Scenarios, Technical Report, 2019.
[4] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, R. Prodan, A serverless real-time data analytics platform for edge computing, IEEE Internet Comput. 21 (4) (2017) 64–71, doi:10.1109/MIC.2017.2911430.
[5] B. Varghese, R. Buyya, Next generation cloud computing: New trends and research directions, Future Gen. Comput. Syst. 79 (2018) 849–861, doi:10.1016/j.future.2017.09.020.
[6] R.F. Hussain, M.A. Salehi, O. Semiari, Serverless Edge Computing for Green Oil and Gas Industry, in: 2019 IEEE Green Technologies Conference(GreenTech), 2019, pp. 1–4, doi:10.1109/greentech.2019.8767119.
[7] 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 Commun. Surv. Tutor. 19 (3) (2017) 1657–1681, doi:10.1109/COMST.2017.2705720.
[8] N. Abbas, Y. Zhang, A. Taherkordi, T. Skeie, Mobile edge computing: a survey, IEEE Internet Things J. 5 (1) (2018) 450–465, doi:10.1109/JIOT.2017.2750180.
[9] P. Porambage, J. Okwuibe, M. Liyanage, M. Ylianttila, T. Taleb, Survey on multi-access edge computing for internet of things realization, IEEE Communications Surveys & Tutorials PP (c) (2018) 1. arXiv: 1805.06695v1
[10] A. Hall, U. Ramachandran, An execution model for serverless functions at the edge, in: Proceedings of the International Conference on Internet of Things Design and Implementation - IoTDI '19, 2019, pp. 225–236, doi:10.1145/3302505.3310084.
[11] A. Madhavapeddy, D.J. Scott, Unikernels: rise of the virtual library operating system, Queue 11 (11) (2013), doi:10.1145/2557963.2566628. 30:30—-30:44
[12] A.G. Tasiopoulos, O. Ascigil, S. Rene, M. Krol, I. Psaras, G. Pavlou, DEEM: enabling microservices via device edge markets, in: 2019 IEEE 20th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM), IEEE, 2019, pp. 1–9, doi:10.1109/WoWMoM.2019.8793004.
[13] 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), 2018-July, IEEE, 2018, pp. 1–11, doi:10.1109/ICCCN.2018.8487326.
[14] S. Sthapit, J. Thompson, N.M. Robertson, J. Hopgood, Computational Load Balancing on the Edge in Absence of Cloud and Fog, IEEE Trans. Mob. Comput. (2018) 1–14, doi:10.1109/TMC.2018.2863301.
[15] X. Liu, Z. Qin, Y. Gao, Resource allocation for edge computing in IoT networks via reinforcement learning, in: ICC 2019 - 2019 IEEE International Conference on Communications (ICC), IEEE, 2019, pp. 1–6, doi:10.1109/ICC.2019.8761385.
[16] S.K. Mohanty, G. Premsankar, M.D. Francesco, An evaluation of open source serverless computing frameworks, IEEE CloudCom, 2018.
[17] 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.
[18] 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.
[19] D. Sabella, S. Antipolis, J. Xhembulla, G. Malnati, P. Torino, S. Scarpina, T.I.M. Telecom, I. Group, A hierarchical MEC architecture : experimenting the RAVEN use-case, in: 2018 IEEE 87th Vehicular Technology Conference (VTC Spring), 2018, pp. 1–5.
[20] E. Schiller, N. Nikaein, E. Kalogeiton, M. Gasparyan, T. Braun, CDS-MEC: NFV/SDN-based application management for MEC in 5G systems, Comput. Netw. 135 (2018) 96–107, doi:10.1016/j.comnet.2018.02.013.
[21] S.C. Huang, Y.C. Luo, B.L. Chen, Y.C. Chung, J. Chou, Application-aware traffic redirection: a mobile edge computing implementation toward future 5G networks 2018-Janua (2018) 17–23, doi:10.1109/SC2.2017.11.
[22] F. Giust, V. Sciancalepore, D. Sabella, M.C. Filippou, S. Mangiante, W. Featherstone, D. Munaretto, Multi-access edge computing: the driver behind the wheel of 5G-connected cars, IEEE Commun. Stand. Mag. 2 (September) (2018) 66–73, doi:10.1109/MCOMSTD.2018.1800013.
[23] M. Król, I. Psaras, NFaaS: named function as a service, in: Proceedings of the 4th ACM Conference on Information-Centric Networking - ICN '17, 11, ACM Press, New York, New York, USA, 2017, pp. 134–144, doi:10.1145/3125719.3125727.

[24] S. Pasteris, S. Wang, M. Herbster, T. He, Service placement with provable guarantees in heterogeneous edge computing systems, in: IEEE INFOCOM 2019 - IEEE Conference on Computer Communications, IEEE, 2019, pp. 514–522, doi:10.1109/INFOCOM.2019.8737449.
[25] S. Meyn, R. Tweedie, Markov Chains and Stochastic Stability, Springer-Verlag, London, 1993. probability.ca/MT
[26] M.I. Naas, J. Boukhobza, P. Raipin Parvedy, L. Lemarchand, An extension to iFogSim to enable the design of data placement strategies, in: 2018 IEEE 2nd International Conference on Fog and Edge Computing (ICFEC), 2018, pp. 1–8, doi:10.1109/CFEC.2018.8358724.
[27] D. Fernández-Cerero, A. Fernández-Montes, A. Jakóbik, J. Kołodziej, M. Toro, SCORE: simulator for cloud optimization of resources and energy consumption, Simul. Model. Pract. Theory 82 (2018) 160–173, doi:10.1016/j.simpat.2018.01.004.
[28] 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.
[29] 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.
[30] P. Beckman, R. Sankaran, C. Catlett, N. Ferrier, R. Jacob, M. Papka, Waggle: an open sensor platform for edge computing, in: 2016 IEEE SENSORS, 2016, pp. 1–3, doi:10.1109/ICSENS.2016.7808975.

Claudio Cicconetti has a PhD in Information Engineering from the University of Pisa (2007), where he also received his Laurea degree in Computer Science En- gineering. He is now a researcher in the Ubiquitous Internet group of IIT-CNR (Italy). He was in the editorial board of Elsevier Computer Networks and has served as a member of the organization committee of several international con- ferences (WoWMoM, IoT-SoS, ISCC, WiOpt, European Wireless, SIMUTools, Valuetools, QoSim, NSTools). He co-authored 50+ papers published in interna- tional journals and peer-reviewed conference proceedings and two international patents.

Marco Conti is the Director of the Institute for Informatics and Telematics of the Italian National Research Council. He has published more than 400 journals and conference proceedings. He is the founding Editor-in-Chief of the Online Social Networks and Media journal and Editor-in-Chief for special issues of the Pervasive and Mobile Computing journal, both published by Elsevier. He has received several awards, including the Best Paper Award at IFIP TC6 Networking 2011, IEEE ISCC 2012 and IEEEWoWMoM 2013. He served as TPC chair for several major conferences, such as IFIP Networking 2002, IEEE WoWMoM 2005, IEEE PerCom 2006, and ACM MobiHoc 2006, and he was general chair (among many others) for IEEE WoWMoM 2006, IEEE MASS 2007 and IEEE PerCom 2010. He is the founder of successful conference and workshop series, such as IEEE AOC, ACM MobiOpp, and IFIP SustainIT.

Andrea Passarella (PhD 2005) is a Research Director at the Institute for Informatics and Telematics (IIT) of the National Research Council of Italy (CNR). Prior to join IIT he was with the Computer Laboratory of the University of Cambridge, UK. He has published 160+ papers on human-centric data management for self-organising networks, Online and Mobile social networks, opportunistic, ad hoc and sensor networks. He received four best paper awards, including at IFIP Networking 2011 and IEEE WoWMoM 2013. He was General Co-Chair for IEEE WoWMoM 2019 and workshops co-chair for IEEE INFOCOM 2019. He was the PC co-chair of IEEE WoWMoM 2011, Workshops co-chair of ACM MobiHoc 2015, IEEE PerCom and WoWMoM 2010, and the co-chair of several IEEE and ACM workshops. He is the founding Associate EiC of the new Elsevier journal Online Social Networks and Media (OSNEM). He is co-author of the book "Online Social Networks: Human Cognitive Constraints in Facebook and Twitter Personal Graphs" (Elsevier, 2015), and was Guest Co-Editor of the special issue Online Social Networks in Elsevier Computer Communications, and several special sections in ACM and Elsevier Journals and of the book “Multi-hop Ad hoc Networks: From Theory to Reality” (2007). He is the chair of the IFIP WG 6.3 “Performance of Communication Systems”.