Toward Distributed Computing Environments with Serverless Solutions in Edge Systems

Claudio Cicconetti, Marco Conti, Andrea Passarella, and Dario Sabella

Computation offloading through stateless applications is gaining momentum thanks to the emergence of serverless frameworks with inherent scalability properties. However, adoption of a serverless framework in an edge computing system requires careful consideration to keep its advantages unscathed.

Abstract

Computation offloading through stateless applications is gaining momentum thanks to the emergence of serverless frameworks with inherent scalability properties. However, adoption of a serverless framework in an edge computing system requires careful consideration to keep its advantages unscathed. In the cloud, micro-services are scaled automatically according to demands, but in edge computing this would incur a significantly higher cost than in a data center and cannot be as fluid. This is especially relevant in scenarios where edge nodes are spread across large areas and have relatively small computation capabilities. In this article we propose to overcome this issue by adapting the allocation of demands to the currently allocated micro-services at short timescales, with two alternative mechanisms designed for different target scenarios, both aimed at enabling distributed computing environments. The proposed solution can be integrated within the ETSI MEC standard, which specifies a reference architecture and open service interfaces. Our contribution is validated in a proof-of-concept scenario with a prototype implementation released as open source.

Introduction

Recently, all major cloud providers (e.g., Amazon, Google, Microsoft) have added to their portfolio a new offer called *serverless computing*, which hides server management from the developers and provides customers with fine-grained billing [1]. The application logic is realized by means of micro-services in a highly scalable infrastructure, as illustrated in Fig 1 (left). Serverless relies on aggressive up-/down-scaling of the application, which makes it difficult to keep a persistent state associated with a running instance. Therefore, function as a service (FaaS) is the most popular programming model for serverless computing: the users specify the operations, called *lambda functions*, to be performed in the requests themselves, either using a high-level language (e.g., Node.js or Python), or onboarding an image from a database under the control of the serverless platform. Every request is stateless, and typically requires very few operations. Load balancing and resource scaling can be implemented effectively using state-of-the-art technology, because all the servers are homogeneous in type and configuration, and they often reside in the same data center.

However, many vertical market segments are becoming increasingly interested in *edge computing* scenarios, where compute nodes are moved in close proximity to the users, in some cases even co-located with the same networking devices providing them with Internet access [2]. Edge computing is a step forward toward the exploitation of computing capabilities distributed across the network, at different levels and in different and heterogeneous entities, often referred to as *distributed computing*. The target scenarios of edge computing are characterized by the possibility to produce and consume flexible services [3], and by heterogeneous workloads and multiple specialized use cases in different business areas, for example, automotive, the Internet of Things (IoT) and industrial automation, virtual reality (VR)/augmented reality (AR), e-health, and smart cities [4]. At a very high level, an edge scenario is represented on the right side of Fig. 1. Deploying FaaS applications on edge nodes, instead of servers in a remote data center, would be extremely beneficial to both users and network operators: the former would enjoy reduced latencies, because edge nodes are geographically closer to the end users, while the latter would experience a cut in outbound network traffic. However, these advantages are not free to take: unlike in a typical serverless deployment, in edge scenarios the nodes performing computation may have much lower capabilities than high-end servers in remote data centers, and they are interconnected by heterogeneous backhaul links, whose capacity in some cases is limited and shared with the whole underlying access network. Thus, it is clear that a general-purpose serverless framework, designed for operation in an elastic virtualization environment, would encounter severe limitations in an edge domain in terms of scalability, performance, and reliability. To take into account by *design* the characteristics of edge computing in the realization of FaaS solutions, in this article we take as a starting point the European Telecommunications Standards Institute (ETSI) Multi-access Edge Computing (MEC) [5], which is a standard designed to address the requirements of several applications in the 5G era by defining a reference architecture and vendor- and application-agnostic interfaces. Then we select from that reference architecture the mechanisms which can be exploited to real-

Digital Object Identifier:
10.1109/MCOM.001.1900498

Claudio Cicconetti (corresponding author), Marco Conti, and Andrea Passarella are with the National Research Council, Pisa, Italy; Dario Sabella is with Intel Deutschland GmbH.

ize a serverless framework, including support for new features required to provide FaaS in distributed edge environments in a way that is transparent to application developers. The proposed solution is presented later and essentially consists of assigning user applications (called contexts in ETSI terminology) to compute nodes, then adjusting such an allocation based on the varying load conditions due to, for example, usage patterns or user mobility. To achieve this goal we present two alternative approaches: changing the service endpoint known to users in a centralized manner (by notifying users) vs. modifying the dispatching of requests inside the edge domain (transparent to users) adopting a *distributed* paradigm. We implemented a prototype to validate the illustrated concept, which also includes initial experimental results to explore the viability of the approach. Conclusions and future areas of investigation on the topic are discussed in the final section.

State of the Art

Serverless computing is a nascent technology. However, it has already attracted significant interest in the research community. The major architectural issues have been examined in [6] as part of the illustration of the project OpenLambda, which is an early enlightening exercise to reproduce a working serverless environment using only open source components. More recently, full-fledged platforms have sprouted, both as commercial solutions and in open source communities (e.g., Knative and Apache OpenWhisk), whose characteristics and performance have been compared in [7] (enterprise) and [8] (open source). In any case, none of the existing solutions has been designed specifically for edge computing, because it creates barriers and limitations in stark contrast with the ever-increasing freedom enjoyed by developers when designing solutions intended for the cloud [9].

As far as the ETSI MEC standard is concerned, in the literature there are some studies that illustrate its reference architecture and objectives. Application loading/unloading is analyzed in [10], where the authors present a proprietary partial implementation of an ETSI MEC system, interacting with an underlying software defined networking (SDN) infrastructure. Interaction between the traffic plane and MEC is also investigated in [11], with the goal of manipulating network operation based on a feedback from edge applications using the Intel©NEV software development kit (SDK) reference platform, today evolved and moved toward OpenNESS (www.openness.org). Finally, integration between SDN and network function virtualization (NFV) is being sought in the EU-funded project 5g-EmPOWER (https://5g-empower.io/), also planning to integrate MEC functions. However, none of the above studies address the specific issues of serverless or distributed computing. In the remainder of this section we introduce the ETSI MEC standard.

ETSI MEC Introduction

An ETSI MEC domain is made of user equipments (UEs) and network-side entities operating on two levels, system and host, as illustrated in Fig. 2. The blueprint also shows the names of the interfaces identified by the standard.

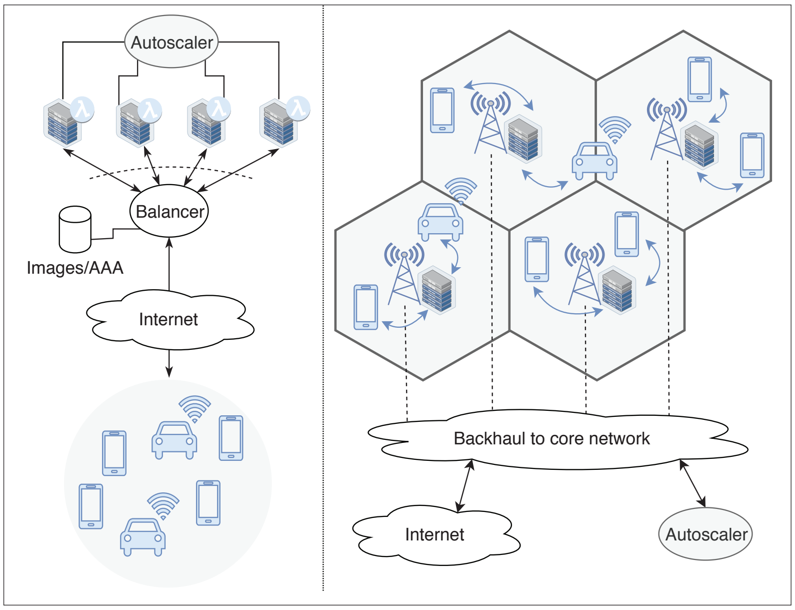


Figure 1. Typical serverless architecture (left) vs. reference edge computing scenario (right).

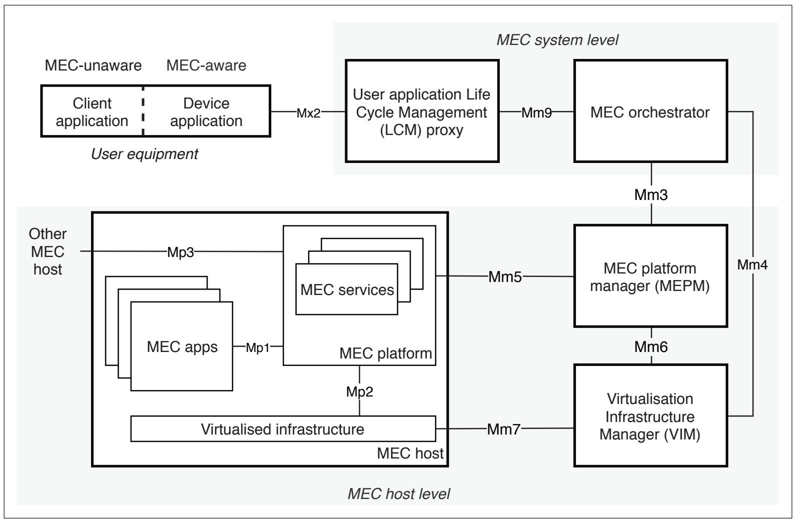


Figure 2. Simplified ETSI MEC architecture.

The UEs are the devices owned and operated directly by the users (e.g., smartphones, IoT sensors/actuators, and connected cars). They consume the services offered by the edge network as *MEC applications*, which are executed by *MEC hosts*: these are edge nodes offering their resources (compute, storage, networking), virtualized at a coarse grain, for example, by means of virtual machines (VMs) or containers together with a reconfigurable virtual interconnecting layer. MEC hosts are distributed over the edge computing domain, ideally in locations that are as close as possible to the UEs (i.e., to the LTE base stations in a mobile access network) to reduce the response time and the outbound traffic. At the system level, the *MEC orchestrator* has a central role, since it is in charge of allocating the resources on the MEC hosts for execution of the applications and services required by the clients, whereas the *user application life cycle management (LCM) proxy* is a mere interface between the UEs and the MEC system.

Once a context has been established, the client and its associated MEC application interact directly via a proprietary interface that is not covered by the ETSI MEC. However, if the MEC orchestrator finds it beneficial or necessary to change the connection of an active context, it may do so by issuing a NotificationEvent to the URI specified by the device application upon context creation.

The MEC hosts are managed by the MEC platform manager (MEPM) and the virtual infrastructure manager (VIM). The former (i.e., the MEPM) is responsible for the configuration of the MEC platform in each MEC host, including managing the life cycles of the MEC applications and services. On the other hand, the VIM (e.g., OpenStack) manages the (virtualized) infrastructure of the MEC hosts. The aspect of virtualization is not directly addressed by the MEC committee and is, in fact, the subject of another group at ETSI, called NFV, which is out of the scope of this work (see, e.g., [12] for an introduction to the subject). For instance, if the MEC orchestrator decides that a new MEC application should be run on a given MEC host, this request is forwarded to the MEPM, which provides the MEC platform with the necessary configuration, while the VM hosting the application is launched and connected by the VIM. Thanks to this clean design and the definition of vendor-neutral interfaces, a network operator may integrate widely used products (e.g., from the Cloud Native Computing Foundation ecosystem), which reduces operational costs.

For all software running in the UE, the standard distinguishes two logical components, with different roles: the *client application* and the *device application*. The client application is the software that implements the application logic with resources on the UE, for example, acquisition from camera and sensors, human-machine interface (HMI), data pre-processing, rendering, and visualization. This component requires that computation and storage are partially realized externally, but it is unaware of where and how such delegation is made possible. On the other hand, the device application is the component responsible for interacting with the MEC system to discover which applications are available in the MEC system and activate a new instance. These operations are done via a REST interface called `Mx2`, using HTTP/1.1 and JSON encoding of resources, introduced below and exploited by our proposed solution:

- A device app may request the list of MEC applications by issuing a `GET` request on `/app_list`; applications are identified by name, provider, version, and so on, and may include minimum quality of service (QoS) requirements.
- Instance activation is done by issuing a `POST` request with an `AppContext` message on `/app_contexts`; the context contains the application identification, as found during the discovery process, and it includes a Uniform Resource Identifier (URI) local to the device application to receive notifications from the MEC system, as seen below. Once the MEC system has provisioned all resources, if necessary, the MEC orchestrator returns an `AppContext` specifying how to actually access the application, for example, an endpoint or URI.

Once a context has been established, the client and its associated MEC application interact directly via a proprietary interface that is not covered by ETSI MEC. However, if the MEC orchestrator finds it beneficial or necessary to change the connection of an active context, it may do so by issuing a NotificationEvent to the URI specified by the device application upon context creation. We exploit this feature of the standard in the next section to optimize resource allocation upon changing conditions.

The `Mp1` interface is defined to allow onboarding of third-party MEC applications into the MEC host, hence fostering an open and prosperous industrial ecosystem. As shown in Fig. 2, MEC hosts run services in addition to applications: they may be used by authorized applications to retrieve additional information from the UE or network status (e.g., the geographical location of users or a date stream for time synchronization) or to influence the edge domain operation for the benefit of the UE application (e.g., to steer traffic more efficiently or to prioritize some traffic flows in the data plane of the access network). The possibility to enhance the serverless framework proposed in the next section by means of MEC services is not explored in this work, but considered of potential interest for future investigation.

Finally, we can expect security concerns to exist in a production system. The standard is not concerned with the interaction between the client and MEC applications, which is outside its reach and view. On the other hand, it specifies all REST commands to be encrypted with Transport Layer Security (TLS) so that no unintended recipient may decode the exchanges, and authorized via OAuth 2.0. According to the latter, the device app obtains a time-limited token from an authentication and authorization service, not addressed in detail in the standard, which grants it permission to issue requests: no exchange is requested per transaction, except for an initial handshake upon the device app first entering the system or to sporadically refresh the token's validity. The ETSI MEC specifications are publicly available (https://www.etsi.org/committee/mec). In this work we refer to version 2.1.1 of the documents released in 2019.

Serverless Edge Computing

In this section we study how to efficiently realize FaaS (i.e., stateless execution of remote functions) in an edge computing system, which we call *serverless edge computing*. Even though our contribution is general, we adopt the reference architecture and terminology of ETSI MEC described earlier to establish a bond with an emerging industry standard. Hence, we aim at implementing FaaS by deploying MEC applications.

Before delving into the matter, we note that any nontrivial application is bound to have some "state," and hence in principle does not adhere to a FaaS paradigm. However, the emergence of serverless computing has shown that there are several real-world applications that can greatly benefit from adopting FaaS, from mobile applications back-ends to complex mathematical calculations, passing through a large variety of IoT services [1, Table 1]. We can assume that for such applications, updating the state, which resides in practice in either the client application or a remote/distributed storage system, does not create a performance bottleneck, and for our purposes in this work can be ignored.

With reference to Fig. 1, in an edge system there can be no centralized front-end balancer, because that would require every lambda function request to go through a component in the core network and then back to the edge nodes, then rendering ineffective having computational resources close to the users. Thus, we assume that every MEC host runs a local serverless framework that includes both the *workers*, that is, the processing units that execute the lambda functions, and a platform for their on-demand activation, all together exposed toward the edge system as a single MEC application (or MEC app, for short). We assume that a single platform/MEC app may serve multiple types of lambda functions, as supported by all major serverless frameworks in the market, and it is in charge of fine-grained scheduling of the incoming requests on the available resources (we consider this aspect outside the scope of our work).

However, auto-scaling is an inherently global function since it must have a system view on the usage of all the MEC hosts and current demands; in the reference ETSI MEC architecture, it could be co-located with the MEC orchestrator. As briefly introduced previously, there are two crucial differences with such an edge deployment and a typical serverless configuration for cloud applications:

- The connectivity between the system-level auto-scaling component and the MEC hosts may be significantly more limited than that in a data center, in terms of latency, bandwidth, and reliability.
- The MEC hosts, in general, have smaller computational capabilities than high-end servers in data centers, which means that they can be controlled at a coarser granularity and cause a higher overhead when allocating/deallocating application images.

Therefore, horizontal scaling cannot be assumed to be as smooth and fast as in a typical serverless deployment. We keep this in mind and describe in the following FaaS operation with ETSI MEC. We start with a baseline strategy, where there is *static assignment* without load balancing, which propels us toward resource optimization solutions based on *centralized assignment*, where client-to-MEC application mapping is updated periodically based on edge system-wide decisions, and *distributed assignment*, where such a mapping is decided based on local measurements only.

Static Assignment

In Fig. 3 (top part) we show an example of a device application creating a context (first `POST` request) after having discovered the available applications with a GET request. Strictly speaking, all the interactions between the device application and the MEC system happen through the `Mx2` interface of the LCM proxy, which, however, is intended as a mere pass-through in both directions: for this reason, in the following we assume that the device application interacts directly with the MEC orchestrator with a slight abuse of terminology. Upon context creation, in addition to saving the newly created context, the MEC orchestrator has to identify the MEC application to be used by the client application sitting behind the device application in the UE. This operation should be performed quickly in order not to delay significantly the start of the serverless operations at the client application. To achieve fast response times even with massive serverless applications, we propose that the MEC orchestrator keeps a simple table where UE and lambda function identifiers (which can also be a wildcard *) are mapped to MEC application service endpoints. The key feature of static assignment is that all the client applications in a given UE are always associated with the same MEC application, and this association remains in place for the whole context's lifetime.

However, in an edge system we expect fast changes due to the small scale of MEC hosts, which are exacerbated if the UEs move at high speed (e.g., cars). This may unbalance temporarily the load *between different MEC hosts* and degrade experience. In medium time windows, such harmful situations can be fixed by global auto-scaling, which is out of the scope of this article. In the short term of an FaaS invocation, we can address this by dispatching function requests to different MEC hosts where the same running images are available. Specifically, in the remainder of this section, we propose to relocate the clients by keeping the same set of running images on the MEC hosts, instead of resorting to dynamic allocation/deallocation of the workers. Note that since we are dealing with stateless applications, this does not require a transfer of the internal state of the application, which is not maintained by any MEC application. Such fine-grained, extremely fast load balancing at the edge is significantly less explored in the literature with respect to global auto-scaling.

Centralized Assignment

Different from the above static assignment, we now explore the relocation of the client-to-MEC application mapping at the edge system level. This can be done in a straightforward manner by using the notification message defined in the ETSI MEC standard, described earlier, as illustrated in the dashed rectangle in Fig. 3. In the example, at some point the MEC orchestrator's table changes, as the result of a periodic edge system optimization process (e.g., see the formal study in [13]), which causes a POST request to be sent to the URI specified by the device application during context creation. Eventually, once the client application terminates its operation, it instructs the device application to send a DELETE command to the MEC orchestrator, which removes the context from the pool of active ones; again, note that this is the only piece of state to be cleared up, since the serving MEC application has none about a specific client application.

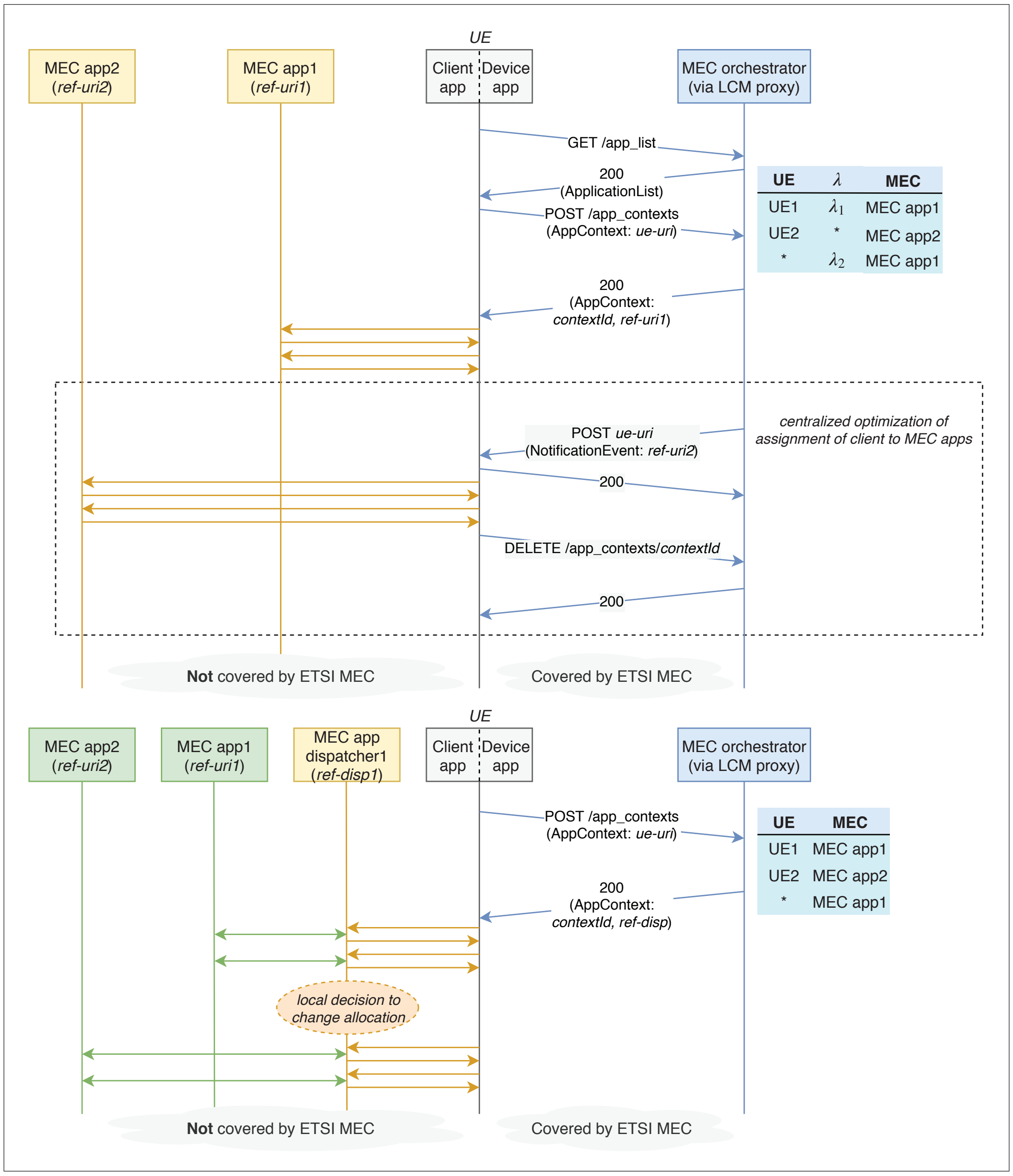
However, this approach has two possible weaknesses depending on the target deployment. First, it relies on system-level optimization, which can be challenging and become a choke point as the edge system size increases. Second, it requires that the device applications host a REST server for the sole purpose of receiving notifications from the MEC orchestrator, which could be unjustified in some cases, for example, if the UE is a constrained IoT device. We address both issues in the following alternative solution.

Distributed Assignment

Finally, we call distributed assignment the dynamic dispatching of lambda functions to the currently most suitable MEC application, trans-

Once a context has been established, the client and its associated MEC application interact directly via a proprietary interface that is not covered by the ETSI MEC. However, if the MEC orchestrator finds it beneficial or necessary to change the connection of an active context it may do so by issuing a NotificationEvent to the URI specified by the device application upon context creation.

By proposing the two approaches here, we intend to raise awareness of the challenges ahead for research and development of solutions in the growing area of serverless edge computing. Further challenges include the study of the integration of ETSI MEC services to exploit estimation of load and mobility patterns, and the revision of generic edge computing optimization models and tools to fully support serverless.



UE
MEC app2 (ref-uri2)
MEC app1 (ref-uri1)
Client app
Device app
MEC orchestrator (via LCM proxy)
GET /app_list
200 (ApplicationList)
POST /app_contexts (AppContext: ue-uri)
UE λ MEC
UE1 λ1 MEC app1
UE2 * MEC app2
* λ2 MEC app1
200 (AppContext: contextId, ref-uri1)
centralized optimization of assignment of client to MEC apps
POST ue-uri (NotificationEvent: ref-uri2)
200
DELETE /app_contexts/contextId
200
Not covered by ETSI MEC
Covered by ETSI MEC
UE
MEC app2 (ref-uri2)
MEC app1 (ref-uri1)
MEC app dispatcher1 (ref-disp1)
Client app
Device app
MEC orchestrator (via LCM proxy)
POST /app_contexts (AppContext: ue-uri)
UE MEC
UE1 MEC app1
UE2 MEC app2
* MEC app1
200 (AppContext: contextId, ref-disp)
local decision to change allocation
Not covered by ETSI MEC
Covered by ETSI MEC

Figure 3. Example sequence diagram.

parent to the client application. We illustrate this approach in the bottom part of Fig. 3, where we added a new type of MEC application, called *dispatcher*: when it receives a lambda function request, it forwards it to the most suitable MEC application in the whole edge system. This way, a device application does not need to be relocated to another MEC host for issues concerning computation load changes. Instead, it is the dispatcher-to-MEC-application connection that is updated, transparent to the client application on the UE. Note that the MEC orchestrator table can be simplified with respect to both static and centralized assignment, since for all lambdas the entry point is always the same MEC application (i.e., the dispatcher). We have already studied the challenge of selecting the best worker in a distributed system in [14], where we propose to use a weighted round-robin scheduler, where the weight is the inverse of an estimate of the network+computational latency for the given worker. This way, a more responsive MEC application will serve a proportionally higher amount of lambda function requests than a slower one. Latency estimates are local to each dispatcher for scalability reasons.

Compared to the centralized assignment, the proposed distributed assignment has three disadvantages:

- Every dispatcher optimizes based on its local (hence myopic) knowledge, which could lead to sub-optimal utilization of resources.
- Dispatchers must keep track of some state associated to all serverless applications in the network, which could be a limiting factor if the MEC hosts have constrained computation.
- Dispatchers act as intermediaries between client and MEC applications; hence, they must be provided off-band all required protocol information and credentials, which might not always be possible.

Performance Evaluation

In this section we show the validity of the assignment approaches above. To this aim, we have realized a prototype implementation of all the system components to execute end-to-end experiments, making available the `Mx2` API as *open source* on GitHub at `ccicconetti/etsimec`. The MEC applications perform face detection using OpenCV via stateless function calls. Evaluation is carried out in a network emulated with Mininet (http://mininet.org/).

The emulated network topology is a fat tree: the core network, hosting the MEC orchestrator, is connected to four MEC hosts via fast backhaul links at 100 Mb/s with 1 μs latency; the access network nodes are arranged in groups of four in "pods," each connected to an MEC host via a slower 25 Mb/s link with 100 μs latency. Further details on the evaluation tool are available in [15].

We ran two experiments aimed at showing the implications of the design choices on performance in limit conditions, described below. In both experiments every UE runs a single client+device application. The system dynamics are deterministic; hence, the performance is expected (and verified a posteriori) to be very stable over multiple repetitions; for this reason, we do not show confidence intervals, and we plot time series. The qualitative behavior described below remains the same by changing the number of clients and MEC hosts, the link speed and latency, and the number of central processing unit (CPU) cores allocated to the MEC applications, although results are not reported here. With static assignment, the orchestrator table is created based on the initial location of UEs and never changed during the experiment, which represents a short time snapshot in between medium-time optimization windows. With centralized assignment, the table is updated every 10 s by the MEC orchestrator running a simple equalization algorithm, which is optimal in the simplified test conditions and evenly spreads the MEC device applications to the available MEC applications.

In *Experiment #1* (slow mobility scenario), starting from a balanced situation with all pods containing the same number of UEs, every 20 s a UE from Pod #1–3 migrates to Pod #0, the final condition being that all UEs are in Pod #0 (hotspot). As can be seen in Fig. 4, with a static assignment the delay curve increases significantly over time, while both dynamic assignment strategies can cope adequately with the load changes.

In Fig. 5 we report the overall network throughput. A static assignment requires much less network traffic than the others because the traffic local to a pod never leaves the corresponding MEC host, while the other strategies sometimes require that clients are served by other MEC hosts. This is especially evident with distributed assignment, since the dispatchers strive to equalize the response delay, which in this case is due almost entirely to the time required for the computation, regardless of the worker's location. Therefore, a trade-off exists between delay and network traffic.

In *Experiment #2* (massive mobility scenario), all the UEs are in one pod; then every 20 s they all migrate to another one. In Fig. 6 we show the

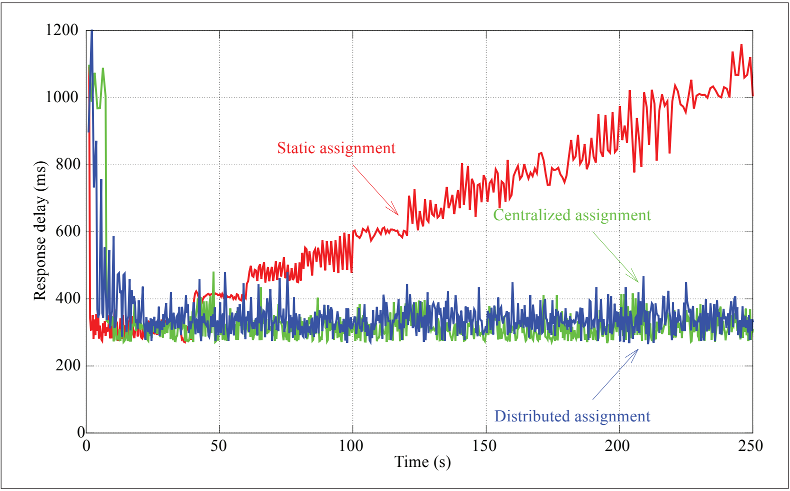


Figure 4. Experiment #1 (every 20 s a UE migrates to a central pod): delay over time.

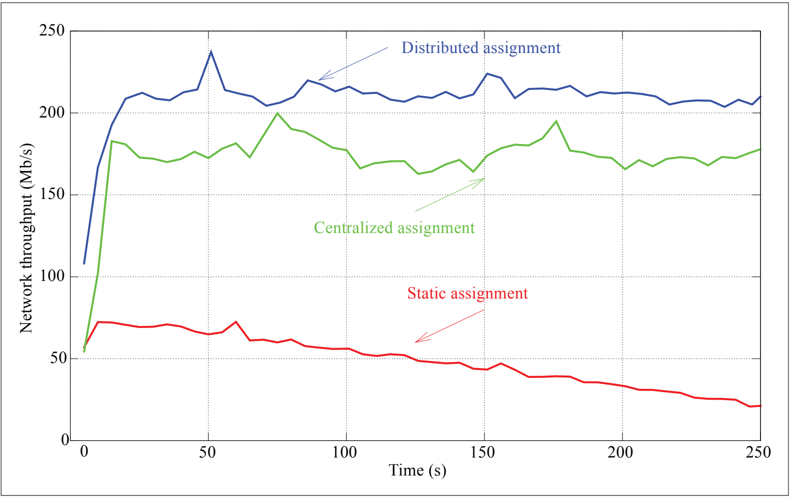


Figure 5. Experiment #1 (every 20 s a UE migrates to a central pod): network load over time.

delay, which is stable with distributed assignment despite the very challenging conditions set by this experiment: only tiny ripples of delays can be noticed at migration times every 20 s. On the other hand, centralized assignment only keeps delay small immediately after an optimization, but they grow significant after each migration, and the static assignment suffers for the whole duration of the experiment.

Conclusions and Future Research Directions

In this article we have provided a tutorial introduction to the ETSI MEC standard, with a specific focus on how it can be exploited to realize serverless edge computing. Furthermore, we have proposed two alternative design approaches to follow fast-changing mobility and load conditions between auto-scaling optimization epochs. With centralized assignment, we notify the UE application via the `Mx2` ETSI MEC interface when a better server is found. With distributed assignment, we delegate the selection of the best server to a dedicated MEC application, called dispatcher, which makes decisions based on local informa-

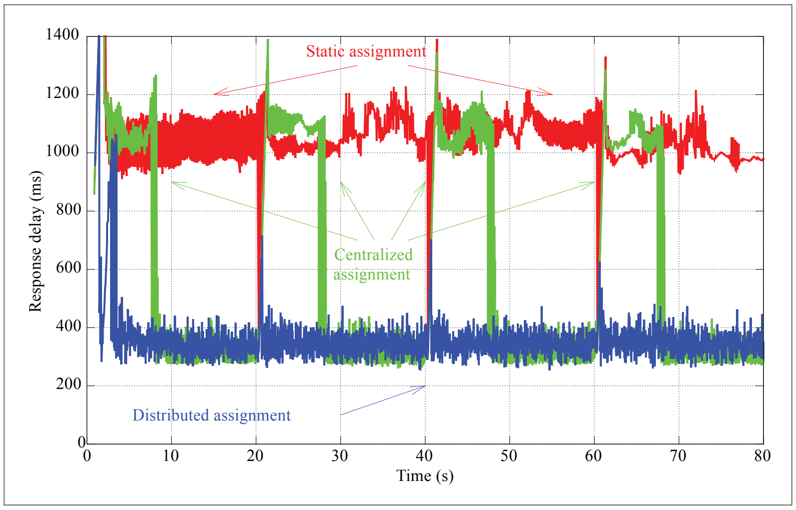


Figure 6. Experiment #2 (every 20 s all UEs in a pod migrate to another): delay over time.

tion only. Using emulation experiments we have shown that both techniques are effective in reacting to system changes.

In general, we argue that a one-solution-fits-all condition does not exist. By proposing the two approaches above, we intend to raise awareness of the challenges ahead for research and development of solutions in the growing area of serverless edge computing. Further challenges include the study of the integration of ETSI MEC services to exploit estimation of load and mobility patterns, and the revision of generic edge computing optimization models and tools to fully support serverless edge computing.

References

[1] P. Castro *et al.*, "The Rise of Serverless Computing," *Commun. ACM*, vol. 62, no. 12, 2019, pp. 44–54.
[2] M. Campbell, "Smart Edge : The Center of Data Gravity Out of the Cloud," *Computer*, vol. 52, Dec. 2019, pp. 99–102.
[3] M. Filippou, D. Sabella, and V. Riccobene, "Flexible MEC Service Consumption Through Edge Host Zoning in 5G Networks," *IEEE WCNC*, 2019.
[4] ETSI," Multi-Access Edge Computing (MEC); Phase 2: Use Cases and Requirements," ETSI GS MEC 002 V2.1.1, 2018.
[5] T. Taleb *et al.*, "On Multi-Access Edge Computing: A Survey of the Emerging 5G Network Edge Cloud Architecture and Orchestration," *IEEE Commun. Surveys & Tutorials*, vol. 19, no. 3, 2017, pp. 165–81.
[6] S. Hendrickson *et al.*, "Serverless Computation with OpenLambda," *USENIX HotCloud*, 2016.
[7] T. Lynn *et al.*, "A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms," *IEEE CloudCom*, 2017.
[8] S. K. Mohanty, G. Premsankar, and M. Di Francesco, "An Evaluation of Open Source Serverless Computing Frameworks," *IEEE CloudCom*, 2018.
[9] S. Venugopal *et al.*, Turn of the Carousel — What Does Edge Computing Change for Distributed Applications?" *ACM ApPLIED*, 2018.
[10] E. Schiller *et al.*, "CDS-MEC: NFV/SDN-Based Application Management for MEC in 5G Systems," *Computer Networks*, vol. 135, 2018, pp. 96–107.
[11] D. Sabella *et al.*, "A Hierarchical MEC Architecture : Experimenting the RAVEN Use-Case," *IEEE VTC-Spring*, 2018.
[12] S. Abdelwahab *et al.*, "Network Function Virtualization in 5G," *IEEE Commun. Mag.*, vol. 54, no. 4, Apr. 2016, pp. 84–91.
[13] L. Wang *et al.*, "MOERA: Mobility-Agnostic Online Resource Allocation for Edge Computing," *IEEE Trans. Mobile Computing*, vol. 18, no. 8, 2018, pp. 1843–56.
[14] C. Cicconetti, M. Conti, and A. Passarella, "An Architectural Framework for Serverless Edge Computing: Design and Emulation Tools," *IEEE CloudCom*, 2018.
[15] C. Cicconetti, M. Conti, and A. Passarella, "Architecture and Performance Evaluation of Distributed Computation Offloading in Edge Computing," *Simulation Modelling Practice and Theory*, 2019.

Biographies

Claudio Cicconetti (Ph.D. Comp. Eng. 07) is a researcher at IIT-CNR, Italy, previously working as an R&D manager at Intecs S.p.a. and a software engineer at MBI S.r.l. He has been involved in several international R&D projects funded by the European Commission and the European Space Agency. He has served as a member of the TPC and organization committees of several international conferences. He has co-authored 60+ papers published in international journals and peer-reviewed conference proceedings and two international patents.

Marco Conti is the director of IIT-CNR. He has published 400+ scientific articles and is the founding Editor-in-Chief of *Online Social Networks and Media*, Editor-in-Chief for Special Issues of *Pervasive and Mobile Computing*, and, for several years, Editor-in-Chief of *Computer Communications*, all published by Elsevier. He has received several awards, including the Best Paper Award at IFIP TC6 Networking 2011, IEEE ISCC 2012, and IEEE WoWMoM 2013. He is the founder of successful conference and workshop series, such as IEEE AOC, ACM MobiOpp, and IFIP SustainIT.

Andrea Passarella (Ph.D. Comp. Eng. '05) is a research director at IIT-CNR, Italy. Previously he was a research associate at the Computer Laboratory, Cambridge, United Kingdom. He has published 150+ papers in peer-reviewed journals and conference proceedings, receiving best paper awards at, among others, IFIP Networking 2011 and IEEE WoWMoM 2013. He has been involved in the organization of several IEEE and ACM workshops, including IEEE WoWMoM 2019 (General Co-Chair) and IEEE INFOCOM 2019 (Workshops Co-Chair), and co-authored the book *Online Social Networks* (Elsevier, 2015). He is the Founding Associate EiC of Elsevier *OSNEM*, and Area Editor for Elsevier *PMC* (best area editor 2019), as well as the Chair of the IFIP TC6 WG 6.3 Performance of Communication Systems.

Dario Sabella works with INTEL as Senior Manager Standards and Research, driving new technologies and edge cloud innovation for advanced systems, involved in ecosystem engagement, and coordinating internal alignment on edge computing across standards and industry groups. Since 2019 he has been Vice-Chairman of ETSI MEC and since 2015 Vice-Chair of IEG WG. He has been a delegate of 5GAA since 2017. He previously worked at Telecom Italia, responsible for research and operational activities on WiMAX, LTE, and 5G. He is an author of several publications (40+) and patents (20+) on wireless communications, energy efficiency, and edge computing. He has also organized several international workshops and conferences.