A Fog Computing Orchestrator Architecture with Service Model Awareness

Gianluca Davoli, *Member, IEEE,* Walter Cerroni, *Senior Member, IEEE,* Davide Borsatti, *Student Member, IEEE,* Mario Valieri, Daniele Tarchi, *Senior Member, IEEE,* and Carla Raffaelli, *Senior Member, IEEE*

***Abstract*—Fog Computing can facilitate the adoption of the Everything-as-a-Service paradigm in infrastructure segments that are located closer to the end user, or to the data source, compared to typical Cloud solutions. This enables combining the advantages of flexible service deployment models with the need to cope with the strict requirements – especially in terms of latency – of emerging applications in softwarized networks. Along comes the need to consider aspects of service orchestration specific to the Fog environment and its intrinsically dynamic nature. In this paper we propose an architecture for flexible Fog Computing service orchestration, with a particular focus on the awareness of service deployment models. We discuss the design choices and describe the components and operations of the proposed orchestration system. We then present a complete working implementation of such architecture, including insights on its ability to handle critical orchestration functions such as service discovery and resource monitoring. We also report on the experimental validation of the system and the performance evaluation on real-world equipment, proving the feasibility and the effectiveness of the approach on a dynamic Fog infrastructure. We complement the work by presenting the results of a combinatorial analysis, validated by simulation, of the service model-aware resource selection process. As a result of our investigation, we show that Fog services can be effectively deployed in a matter of a few seconds, or even in less than one second when suitable Fog nodes are available, taking advantage of the awareness of the available service models.**

***Index Terms*—Fog Computing, service orchestration, service deployment, XaaS, experimental testbed**

I. Introduction

THE new generation of communication infrastructures brings about radical changes, with progressively increasing network *softwarization* enabling unprecedented levels of pervasiveness, flexibility, adaptability and automation. This trend is characterized by increasingly critical needs in terms of networking and computing performance, depending on application context [1]. New generation network services must in many cases ensure very tight latency values experienced by end-user or machine-to-machine applications, thus making latency one of the most important parameters in the performance evaluation process of the service provider. Concurrently, the pervasiveness of Internet of Things (IoT) devices requires highly scalable approaches to massive data processing and storage [2].

G. Davoli, W. Cerroni, D. Borsatti, D. Tarchi and C. Raffaelli are with the Department of Electrical, Electronic and Information Engineering "G. Marconi", University of Bologna, Viale del Risorgimento, 2, 40136, Bologna, Italy. E-mail: {first_name}.{last_name}@unibo.it

M. Valieri contributed to this work as a graduate student in Telecommunications Engineering at the University of Bologna, Italy. Currently, he is with Eggtronic, Italy. E-mail: mario.valieri@studio.unibo.it

In the last decade, the features offered by the Cloud Computing paradigm obtained great popularity among telco operators, and the Cloud approach has been delegated the majority of networking and computing functionalities, including how end-user applications and vertical services are deployed. However, some of those cloud-based applications and services may not need to perform burdensome calculations, but are rather interested in achieving the lowest possible latency, even at the expense of computing power. This is especially true for Cloud applications based on microservice design, that are widespread in *softwarized* networks due to their highly scalable nature and very efficient lifecycle manageability [3]. This kind of applications could benefit from the presence of suitable devices located closer to the end user or to the source of IoT data with respect to centralized Cloud infrastructures, or even located at the edge of the network. Such devices can be used for remote processing, computation *offloading* operations, resource relocation, and service deployment, with significantly reduced latency and improved scalability.

In this scenario, Fog Computing proposes an intermediate layer between end users and Cloud infrastructures [4], where clusters of Fog nodes provide services that are similar to those offered by its larger counterpart (i.e., Cloud Computing) but focusing on the needs of microservice-based and modular applications [5], [6]. In addition to tackling latency, the Fog Computing paradigm operates on ecosystems that are inherently dynamic, as nodes offering computation power may join or remove themselves from a Fog cluster, making the overall amount of available resources in the cluster varying over time [7]–[9]. Moreover, the capabilities and the amount of resources offered by a single Fog node may not be known a priori, i.e., before the node itself joins the Fog cluster.

Although such distributed and dynamic characteristics make Fog infrastructures different from traditional Cloud Computing ones, the services offered can still be considered similar [4]. Hence, it is quite natural to borrow the Everything-as-a-Service (XaaS) Cloud service model classification [10] and apply it to Fog Computing infrastructures as well, thus extending it towards a more flexible and dynamic scenario. While the reuse of XaaS models from the Cloud Computing scenario allows to profit from their specific characteristics, **an additional effort is required to redefine the usage of those models in a dynamic environment where resource-constrained nodes are employed** to provide one out of a set of supported services, based on their availability at the time when a user requests it. In other words, this introduces **the need to design a suitable Fog Computing architecture that**

is aware of different service models and related resource monitoring and allocation issues. At the very heart of the Fog architecture, **a compatible service orchestration system needs to be designed and developed to take advantage of the different service models supported by the Fog infrastructure**.

In particular, the orchestrator should be aware whether a given service can be offered natively by a Fog node, or can be deployed by properly customizing or programming a specific platform or a more general execution environment available on a Fog node. The challenges that such an orchestrator must address include the discovery of how many and which kinds of Fog nodes are available, the knowledge of the service models the Fog nodes support, and the constant monitoring of the current status of the Fog node resources. Additionally, recognizing that a service may be composed of multiple micro-services, the orchestrator should also retain the knowledge of which micro-services are needed and how to properly activate them in order to offer a composite service, which includes facing precedence and interdependence issues among service components. Similar considerations apply to the case of distributed services, such as those that need to process parallel data streams [11], therefore requiring the activation of multiple coordinated service instances.

Considering all of the above aspects, the orchestration system will then be capable of handling service requests coming from end-users or machine-to-machine applications (i.e., the *service consumers*), and choosing which node, if any, can be employed to provide the requested service.[1] Finally, this Fog orchestrator will proceed to configure the node for the task, and inform the consumer about the availability of the requested service.

Based on the considerations expressed above, an original Fog Computing system architecture aware of different XaaS service models is here proposed, and designed to administer service activation in a generic, distributed, and dynamic Fog infrastructure. As a key functionality of the proposed architecture, we designed and developed a modular Fog service orchestration system that is able to act as a resource management and service provisioning layer placed between service consumers and the Fog infrastructure. Although we already demonstrated the main concept of a service model-aware Fog orchestrator in a preliminary paper [12], the system architecture and the components presented here were completely redesigned to enable a more efficient modular implementation. In addition, the new architecture and the related implementation – available for download at [13] – were experimentally validated on a suitable testbed, comprising both virtual and physical devices, whereas the flexibility of the service model-aware Fog node selection approach was quantitatively evaluated through combinatorial analysis.

[1]In this paper we use the term *service request* to refer to a management plane interaction between consumers and orchestrator, i.e. the request made by a consumer to the orchestrator to discover the availability or to activate a given service. Differently, we use the term *service consumption* to refer to a data plane interaction between consumers and Fog nodes, i.e. the request made by a consumer to a Fog node, where the requested service is running, to obtain the requested service.

Some of the architectural choices made in our work were naturally inspired by the Multi-access Edge Computing (MEC) framework proposed by the European Telecommunications Standards Institute (ETSI) [14]. However, we would like to clarify that the scope of the Fog architecture proposed in this paper is intended to be wider than the one considered by MEC, under a number of aspects. For instance, MEC relies on a Cloud-like infrastructure located at the network edge, where resources are represented by nodes usually residing in a datacenter-like facility. In addition to that, in our view Fog clusters may also comprise relatively smaller and sparse nodes connected to the network infrastructure, located at the access or between the edge and the cloud. Fog nodes can even be part of the network infrastructure itself, such as wireless access points, customer appliances, or other embedded computing devices, whose resources can be harnessed for Fog service activation. Moreover, MEC applications are typically designed to be deployed as virtualized entities, whereas the proposed Fog system may also make use of services that are natively offered by devices that do not support virtualization. Although there is an ongoing discussion within the MEC standardization group to consider also constrained devices and non-virtualized platforms to run MEC applications, specific standards and a working implementation of a MEC platform supporting such a wider scope are not currently available, to the best of our knowledge. This difference in scope, however, can be leveraged in terms of complementarity between the two architectures, combining MEC principles with those followed by the system presented in this paper. Interoperability can be achieved by adhering to a concerted set of interfaces, obtaining a framework for the activation of services over a wide environment, as shown in [15].

In the following sections, we first discuss some relevant related work, then clarify what supporting multiple Fog service models means in the context of this work. We then describe the proposed architecture, detailing its components and their functionalities, as well as the intended procedures through which it can orchestrate service provisioning in the described service model-aware scenario. We consequently introduce an original implementation of the devised Fog Computing system. A combinatorial analysis of the service model-aware Fog resource selection process is formulated, followed by a set of performance evaluation tests, most of which carried out on real-world equipment. Main achievements, ongoing research, and future directions are presented as concluding remarks.

II. Related Work

Since the introduction of the Fog Computing paradigm, there has been a growing interest in the research community to identify related suitable system architectures. In [16], IEEE started a standardization effort for defining a reference architecture able to distribute the computing, storage, network, and control functions in a Cloud-to-Things continuum. However, to the best of our knowledge, no practical implementations have been proposed based on the IEEE recommendations. Additionally, the conceptual model of Fog Computing and how it relates to Cloud Computing was recently reported by

the National Institute of Standards and Technology (NIST) [4]. In particular, the NIST recommendation defines three types of Fog node architectural service models, namely Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS), which replicate the classification previously adopted for Cloud Computing service models [10].

Development efforts have been carried out by private companies as well, proposing a variety of Fog Computing architectures, often applying context-specific or proprietary approaches, tailored for the sale model and market segment they intend to target [17].

A large portion of past and ongoing work focuses on the needs of IoT devices and services they can offer [18], often specifically to the industrial environment [19], [20]. An assortment of research efforts, including [21], [22], target a wide range of issues of service provisioning in Fog scenarios, including but not limited to heterogeneity of resources, scheduling, and resilience. For instance, [23] tackles the challenge of Fog service scheduling with a highly scalable orchestration system, where host nodes determine relevant parameters and distances to each other, and rely on Cloud processing for the final decision on allocation. In addition to those, another research trend is devoted to integration of new ecosystems at the edge of the network on which to deploy services [24], [25], fostering an end-to-end approach to service provisioning in a holistic perspective [26]. However, the initiatives mentioned above do not consider the possibility and the consequent advantages of applying an XaaS-like model to service deployment, mostly due to trading flexibility for performance.

An additional important research topic revolves around service placement and its challenges. These issues applied to a Fog scenario are the focus of [27], where an algorithm to optimise the placement of applications modules is presented. In [28], challenges stemming from the usage of heterogeneous constrained devices are addressed, with a focus on efficient service placement over available Fog nodes. We assert that *service placement* and *service orchestration* are two different tasks with different scopes. The former is concerned with the optimization of the choice of where to run a certain service, but it does not consider aspects such as resource discovery or monitoring, as the latter does. Therefore our work is to be considered as an orchestration effort, as it covers more aspects than just service placement, including aspects related to service lifecycle management.

In the direction of "as-a-service" provisioning, in [29] the concept of "Fog-as-a-Service" is explored, by proposing a Fog service model for massive IoT data in smart cities and offering the whole Fog service orchestration system "as-a-service". This interpretation differs from the XaaS-aware approach proposed here, which contemplates achieving dynamically adaptive service orchestration by applying different models to the provisioning of individual services, rather than offering the whole orchestration system as a Cloud-like service. This results in a more flexible and general-purpose solution intended for orchestrating any kind of Fog Computing node.

Referring to the XaaS classification, works such as [30] start investigating the benefits of flexible deployment of services over Fog infrastructures, specifically adopting containerized solutions, yielding promising results. The usage of containers in Fog environments and related orchestration needs are further investigated in [31], where an optimization strategy for scheduling of containers is proposed, but without considering XaaS-awareness. Practical aspects of containerized solutions are addressed in [32], where a strategy for proactive caching of container images is presented and validated with numerical simulations.

In [33] the authors propose how to provide a Fog computing environment using consumer networking devices. Computing power comes from the networking devices themselves, or the system may harness it from other devices connected to networking equipment. This work, while applying the XaaS model to service deployment in the contemplated scenario, includes fixed hardware only. Differently, the system here proposed enables harnessing computing power from devices that may dynamically and unexpectedly connect to the infrastructure.

Moving towards XaaS-awareness, in [34] the authors propose an optimization framework for on-demand service deployment in a XaaS Fog Computing environment, where a resource-constrained latency minimization approach is considered. The system has been studied through numerical modeling, although no practical implementation has been considered. Furthermore, [35] proposes an application deployment strategy with a service-agnostic approach. A peer-to-peer Fog-based computation offloading service is proposed in [36], where multiple Android-based nodes exchange parameters and source code, thus allowing to implement the computation offloading service using either the SaaS or the PaaS model.

To the best of our knowledge, no previous work tackles the challenge of XaaS-aware service orchestration on Fog Computing scenarios in the way we propose to do it. Building on the lessons learned from the research efforts mentioned above, and evolving our first demonstration of a service model-aware Fog orchestrator [12], we designed, developed, and implemented the system presented in the following sections.

III. Fog Computing Service Models

As argued in the previous sections, a Fog Computing environment can be regarded as complementary to a traditional Cloud Computing one, facilitating the definition of a new distributed scenario. This is often referred to as Edge-to-Cloud continuum [37] or Cloud-to-Thing continuum [38], where the processing capabilities are distributed along the path from the service consumer to the Cloud [39]. With this in mind, a Fog Computing architecture should be able to map the same traditional Cloud Computing XaaS service models, i.e., SaaS, PaaS, and IaaS categories, into an on-demand, resource-constrained distributed environment.

In a Cloud Computing environment, the **SaaS** model allows to access a software application instance running on a remote server through a predefined interface. The same approach can be extended to a Fog Computing architecture, where a Fog node is hosting a *specific* application that can be accessed

by any consumer through a *specific* interface. Similarly to the Cloud architecture, the consumer may only request the already-deployed applications, benefiting from the lowest possible service activation time [34]. The data exchanged between consumers and nodes is limited to the input parameters and output results.

The **PaaS** model, instead, allows the consumer to access a remote platform or operating system instance, allowing the development of custom applications, by using programming languages, libraries, and tools provided by the platform itself. When extended to the Fog environment, the PaaS model allows the node to provide a *generic* application that can be deployed or developed through a *specific* set of libraries, platforms, or programming languages. Within this scenario, a consumer can interact with the Fog orchestration components for deploying the source code of an application and have it compiled and executed on the PaaS node. In this case, the amount of exchanged data also includes the code for the application to be executed, and a higher service activation time is expected, with reference to the SaaS case.

Lastly, the **IaaS** model provides the consumer with access to virtualized computing, storage, and networking resources for installing and running a completely custom system. When extended to the Fog environment, this model allows implementing a service where a Fog node is able to provide a *generic* application on a *generic* platform through a programmable infrastructure (e.g., the virtualization environment). In this case, the service to be activated must be provided in a way that is compatible with the virtualization environment offered by the Fog node, e.g., a container management/orchestration platform such as Docker or Kubernetes, a virtual machine hypervisor such as KVM or VMware, other forms of lightweight virtualization such as Unikernels, etc. In this way, a virtualized instance can be correctly deployed and run on the hosting node, implementing the required service. In this scenario, the service activation process may include additional steps of interaction between consumer and node, as well as the download of the virtual components needed to deploy the service. Hence, a repository should be foreseen for storing available images to be used by a Fog node upon request.

It is worth noting that the extension of the different Cloud service models to a Fog environment retains similar characteristics, because when moving from the SaaS through the PaaS to the IaaS model, we move from the lowest to the highest degree of generality and flexibility, while simultaneously moving from the fastest to the slowest service activation time. **In order to take advantage of the different levels of flexibility and consider the different activation times offered by different service models, we argue that a Fog Computing orchestration system should be aware of how a given service can be deployed, not only in terms of which Fog node resources must be used, but also according to which model it can be deployed.**

IV. Fog Computing System Architecture

Developing a Fog Computing system for XaaS-aware service provisioning requires the design of a suitable reference architecture, which is able to represent various aspects of service orchestration, including consumer interaction, decision making, monitoring, control, and management processes, properly extended to support the inherent flexibility that different service models can offer.

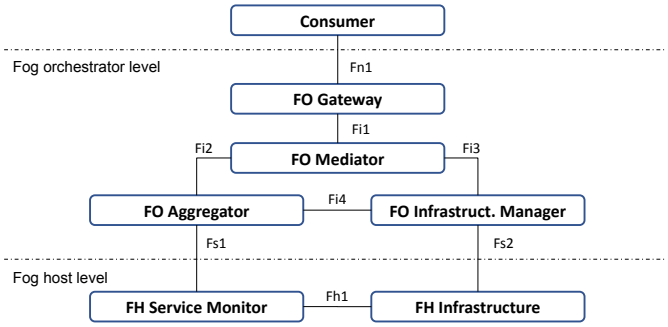


Fig. 1. Reference architecture of the proposed service model-aware Fog Computing system, including the Fog orchestration (FO) level and the Fog host (FH) level. The functional components of the architecture are shown, including the reference points used for relevant interactions.

The envisioned architecture is depicted in Fig. 1, and can be logically divided into two levels. The *Fog orchestration level* contains the logically-centralized intelligence of the system, including decision-making components and data-collection modules. Elements of this level are logically unique in a deployment of the Fog orchestration system, although their instances might be replicated for scalability. On the other hand, the *Fog host level* contains the elements that are needed on any Fog host node to support the interaction with the Fog orchestration system. Therefore, these elements are meant to be instantiated on every Fog host node, with each instance being logically independent of other instances running on other nodes. The hierarchical structure of the architecture offers a logical separation that helps in making the system flexible and capable of supporting a variety of infrastructures and nodes to be employed for service provisioning.

As previously mentioned, the reader can find some commonalities between the proposed Fog system architecture and the MEC architecture standardized by ETSI [14]. However, the strong point of the proposed architecture resides in the native high flexibility in terms of Fog host nodes that can be included in the system and employed for service activation. In a MEC scenario, services are typically activated as virtual components over an edge-cloud infrastructure, whereas here, thanks to the service model awareness, services can be deployed in different ways depending on the specific nature of the Fog host node, e.g., instantiated as virtual components or allocated on physical nodes that offer that service natively. In fact, the *Fs2* reference point, utilized to manage the activation of services on a Fog host node, is not specific to a given technology, but it can be used to interact both with physical and virtual equipment. This approach allows to include resources dynamically, a feature that can offer great help in emergency scenarios, where not only latency but also availability and reliability are crucial.

The remainder of this section briefly introduces the role of each functional component, whereas the specific interaction mechanisms between components and the purpose of the

reference points are discussed in the next section. For brevity, the name of components belonging to the Fog orchestration level is prefixed by the acronym "FO", while components residing in the Fog host level have their name prefixed by "FH".

A. Fog orchestration level

Service requests are received by the *FO Gateway*, which offers the point of contact for service consumers to interact with the orchestration platform, defining a suitable level of service abstraction. It also provides authentication and syntax validation of service requests.

The most crucial component of the architecture is the *FO Mediator*, which represents the core functionality of the orchestration system. It is in charge of maintaining an overall view of the Fog ecosystem, processing service requests, and making decisions on service activation and service model to be used, based on current service availability and resource status information gathered from the underlying infrastructure through interaction with the monitoring modules and agents of the system.

Such information on the infrastructure status is collected by the *FO Aggregator*, whose objective is to combine monitoring data about the infrastructure and expose it for consumption by the FO Mediator, which makes use of this data for service model selection and service activation. The FO Aggregator is also directly in charge of gathering information on available services, by interacting with the relevant component on Fog host nodes.

Finally, the *FO Infrastructure Manager* oversees the activation of services on Fog host nodes, according to the decisions made by the FO Mediator. The FO Infrastructure Manager is responsible for dealing with technology-specific details depending on the particular computing platform used by Fog host nodes. It is also in charge of collecting information on Fog host node resource utilization and passing them to the FO Aggregator for preservation. Therefore, the FO Infrastructure Manager should be equipped with multiple "adapters" to interact with heterogeneous Fog Computing node implementation technologies, or multiple FO Infrastructure Managers should be deployed. It is worth noticing that this modular approach, with the definition of abstracted interfaces between functional elements, allows the system to be extended also to hybrid infrastructures, where multiple "computing sites" could be available (e.g., federated Fog sites connected to a central Cloud) and a FO Infrastructure Manager could be deployed for each of them. This would not impact the orchestration operations, which would then take the specific characteristic of each site (e.g., latency) into account for the decisions on service activation.

B. Fog host level

The *FH Service Monitor* is a component located on each Fog host node, and it is in charge of monitoring the availability of services on the node, registering the services that are available when the node connects to the Fog cluster, and keeping track of their activation. It supplies the FO Aggregator with this information, which inherently includes the indication on the service model(s) this node supports.

The *FH Infrastructure* represents the collection of all hardware and software elements that build up the Fog Computing nodes where services can be activated and executed. In general, the FH Infrastructure can span across several locations, from the cloud to the edge, to the access, up to the end-user premises. It includes the specific management/control platforms and interfaces that can be used by the FO Infrastructure Manager to communicate with it and activate services. It also gathers information on hardware resource utilization within the Fog node (e.g., CPU utilization, residual disk space, etc.) for collection by the FO Infrastructure Manager. The features offered by the FH Infrastructure determine the way the orchestration system can handle scaling and service availability. If the FH Infrastructure has native support for such features (e.g., in case of a Kubernetes cluster), then the FO Infrastructure Manager can simply instruct it by means of high-level policies specified according to the given FH Infrastructure management interface. Conversely, the FO Mediator can take advantage of the view it has over the available resources to include the activation of redundant services in the set of orchestration decisions it is responsible for, as well as reacting to an excessive usage of such resources, and oversee scaling operations accordingly.

V. Fog Service Orchestration Operations

The functional components introduced in Section IV interact according to predefined patterns and procedures. Before describing how they operate when a new service request is received, we need to define and clarify some relevant properties and functions.

A. Properties and functions

1) Service categories: According to the three Fog service models described in Section III, we distinguish three types of software entities that may run on Fog host nodes:

- *Applications* (APPs): they represent service instances natively offered by SaaS nodes. An APP may take values as input and return results based on those values (e.g., in case an APP performs a computationally-intensive series of operations on input data), or it may listen for incoming requests and serve them (e.g., in case of a Web-based application).
- *Software Development Platforms* (SDPs): they represent service development environments natively offered by PaaS nodes (e.g., Remote Java or Python interpreters). A SDP takes as input blocks of source code written in a predetermined language and/or using specific development libraries, executes them and returns any output to the consumer.
- *Fog Virtualization Engines* (FVEs): they represent general-purpose virtualization environments available on IaaS nodes (e.g., Docker, Kubernetes, KVM, VMware). A FVE enables a Fog host node to instantiate virtualized appliances based on existing or customized images, giving the consumer maximum flexibility to deploy any

required computing environment and use it to perform any task.

To better clarify the distinction, we can consider the example of a virtual network function (VNF) performing video transcoding. Such VNF could be offered as an APP (i.e., a monolithic entity that is able to operate on video streams using a predefined codec), as well as a SDP (i.e., an implementation that allows the service consumer to upload the desired codec library, among a set of supported ones, before operating on the video stream), and a FVE (i.e., an environment supporting the deployment of the entire transcoding engine and related codecs to be used). It is worth remarking that the proposed system is not limited to the activation of VNFs, but it also supports native applications, or services disaggregated into multiple microservices.

In the following, the term *service* represents either an APP, a SDP or a FVE, regardless of the technology that the infrastructure employs to provide that service to the consumer. For instance, if a consumer asks for a computation offloading service, that service will be provided either as an APP natively running on a SaaS node if available, or as a virtualized instance of that APP deployed using a FVE on an IaaS node. In other cases, the consumer may deliberately ask for a SDP or a FVE as the intended service, but this is supported mainly for generality reasons, as we do not envision this to be the norm. In any case, based on the current service availability within the FH Infrastructure, the Fog orchestration system is able to decide on the most suitable service model to execute the requested task.

2) Service activation: In generic terms, a service is *activated* (opposite: *deactivated*) when a consumer sends a request for that service and the request can be satisfied. If the requested service is natively available on a node (e.g., if the consumer requests an APP/SDP/FVE that is running on a SaaS/PaaS/IaaS node), then it can be *allocated* (opposite: *deallocated*) to the requesting consumer. Conversely, if the service is not natively available and should be provided by means of an additional instantiation step (e.g., by launching a container in order to provide an APP/SDP on an IaaS node), then the service needs to be *deployed* (opposite: *destroyed*).

3) Service discovery: Each Fog host node offers a certain set of services and is capable of providing a certain set of service models. Each service is associated to an identifier that must be unique within the overall set of services offered in the Fog ecosystem, but the same service may be offered by multiple host nodes. Each Fog host node announces the set of services it offers to the relevant orchestration-level entity, which is thus inherently made aware of the service models provided by that Fog host node. Orchestrator and hosts must share the same common knowledge over the set of existing and available services. This may be accomplished either via custom solutions, or by relying on existing protocols for service discovery.

4) Service images: For the purpose of this work, the term *service image* refers to a virtual component that, when deployed, is able to instantiate a certain service. The orchestrator maintains a list of service image repositories, which can be pre-configured before startup and then updated at runtime. For instance, service images include – but are not limited to – container images, which represent a typical example of an executable package of software that includes everything needed to run a service. The Fog service orchestration system may also support service image on-boarding functions, a choice that could have several advantages in terms of efficiency and security. Although the approach we propose here intends to be general, we cannot expect that all possible services are available in all possible formats. However, if a service is not available natively, the orchestrator looks for, e.g., the image of a containerized version of the same service, if it exists in a public or private repository. This is not too unrealistic if we consider, for instance, the popularity of the Docker Hub repository and the containerized version of many applications available there, often shared by third-party developers.

B. Processing of service requests

Service consumers (e.g., Operations Support System, third-party verticals, external applications) formulate requests that trigger the discovery or activation of services in the Fog system. These requests include only essential information on the desired service, with the only mandatory value being the unambiguous service identifier, taken from a set of known service names, and possibly also comprising a list of constraints and requirements, taken from a predefined set. When activating a service on a Fog host node, the orchestrator takes a number of aspects into account, including but not limited to the deployment model of the service and the constraints and requirements specified in the request.

Figures 2 to 4 depict the sequence of interactions among functional elements of the system triggered in reaction to different requests coming from the consumer. Specifically, Fig. 2 focuses on the case where the consumer requests the list of currently available services, while Figs. 3 and 4 represent the way the system handles the request for the activation of a new service, in case the service is available and can therefore be allocated, or in case the service is not available but can be deployed, respectively.

For any request type, the consumer (C) interacts with the service orchestrator reaching the FO Gateway (FO-GW) through the *Fn1* reference point, which represents the northbound interface of our Fog orchestration system. The FO Gateway authenticates the consumer and validates the correct syntax of the request. The request is then passed, through the *Fi1* reference point, to the FO Mediator (FO-M), which interacts with the FO Aggregator (FO-A), via the *Fi2* reference point, to gather information on the available services offered by the set of Fog host nodes. The FO-A, in turn, interacts with the FH Service Monitor (FH-SM) of all available Fog host nodes, querying for the current availability of services. If the consumer has only requested the list of currently available services, as shown in Fig. 2, the procedure ends here, with the cumulative response being routed back to the consumer through the FO Gateway.

However, if the request involves the activation of a service, the FO-M relies on the information on currently available services and resource utilization to determine whether and

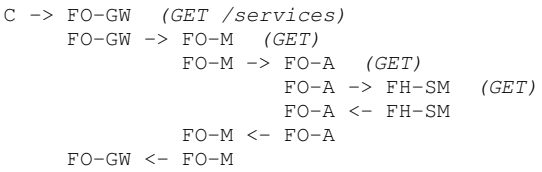


Fig. 2. Sequence diagram of the interactions in case the consumer requests the list of available Fog services. The methods displayed with *slanted text* refer to the REST operation through which the interaction was implemented (see Section VI-C).

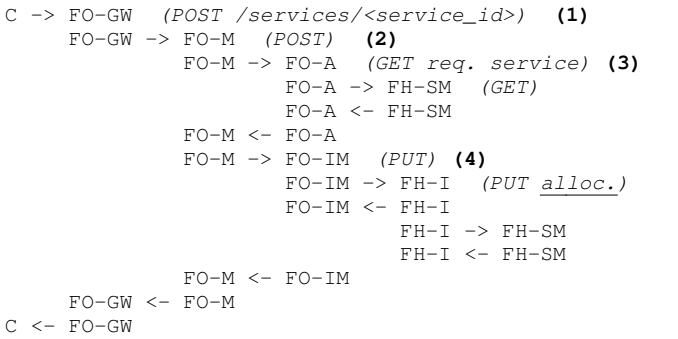


Fig. 3. Sequence diagram of the interactions in case the consumer requests a service that can be activated via simple *allocation*. The methods displayed with *slanted text* refer to the REST operation through which the interaction was implemented, while numbers in **bold text** are used as references in the performance evaluation (see Section VIII-B).

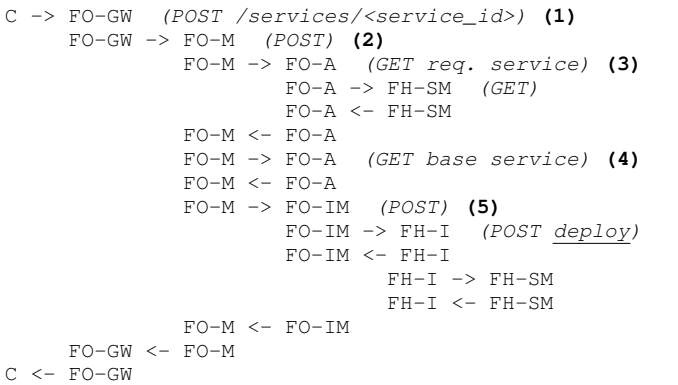


Fig. 4. Sequence diagram of the interactions in case the consumer requests a service that must be activated through *deployment* on IaaS nodes. The methods displayed with *slanted text* refer to the REST operation through which the interaction was implemented, while numbers in **bold text** are used as references in the performance evaluation.

how to activate the requested service. The FO-M tries to activate the service in the most efficient way possible, i.e., by picking the most convenient service model and host node. For a host node to be eligible to run the service, its resource utilization must not exceed a certain threshold. Both the specific resource utilization parameter to be considered for this selection and the threshold to be applied on it are a matter for the initial configuration of the orchestrator (e.g., a node may be considered eligible only if its CPU utilization does not exceed 90%). If multiple nodes are eligible, any optimal choice algorithm can be applied in order to make sure that the best node is selected, based on specific criteria [34].

Depending on the service category being requested and the software entities available on eligible Fog host nodes, the service activation involves different interactions. If the consumer requests a given APP, the FO-M scans the list, retrieved from the FO-A, of available host nodes natively providing that APP, trying to give priority to the SaaS model and minimize the activation time. It then discards nodes whose utilization exceeds the threshold. If at least one eligible SaaS node is available, then the FO-M chooses the best node to *allocate* the service on, and proceeds by instructing the FO-IM about the allocation via the reference point *Fi3*, as depicted in the sequence diagram in Fig. 3. The FO-IM interacts with the FH Infrastructure (FH-I) of the chosen Fog host node to configure the access to the service by enabling the consumer who requested it. The FH-I also communicates the new status of the resources to the FH-SM located within the same node through the reference point *Fh1*. If the allocation succeeds, details on how to reach the node and the allocated service running on it are sent back to the consumer in the final interaction through Fn1. In case the service requested by the consumer is a SDP or a FVE, the procedure is similar to the one described above, with the difference that the FO-M searches for eligible Fog nodes only among PaaS and IaaS nodes, respectively.

On the other hand, if a service cannot be natively allocated, because an eligible SaaS or PaaS node is not available to run the requested APP or SDP, respectively, the FO-M tries to *deploy* it on a IaaS node. In order to do so, the FO-M looks up the requested service in the list of available service images (as defined in Section V-A4). If a suitable service image is found, the FO-M follows the procedure to search for an available IaaS host node offering the required *base service*, i.e., the FVE on top of which the service requested by the consumer must be deployed, as illustrated in Fig. 4. If an eligible node is found, the service is deployed on it by triggering the deployment through the FO-IM via the *Fi3* reference point. The FO-IM in turn interacts with the FH-I via the *Fs2* reference point, deploying the service on the Fog host node. Information on the eventually deployed service is then included by the FO-M in the response to the consumer at the end of the sequence diagram.

In case a requested APP or SDP cannot be natively allocated on a SaaS or PaaS node, nor deployed on an IaaS node, or in case a requested FVE is not available on any IaaS node, then the service activation procedure fails and the system replies to the consumer with a message indicating failure in activating the service.

VI. Fog Orchestration System Implementation

Based on the reference architecture presented in Section IV, we implemented a system we refer to as FORCH (a system for Fog ORCHestration) [13], which covers all aspects of service model-aware orchestration discussed in Section V. FORCH coordinates the activities in the Fog Computing

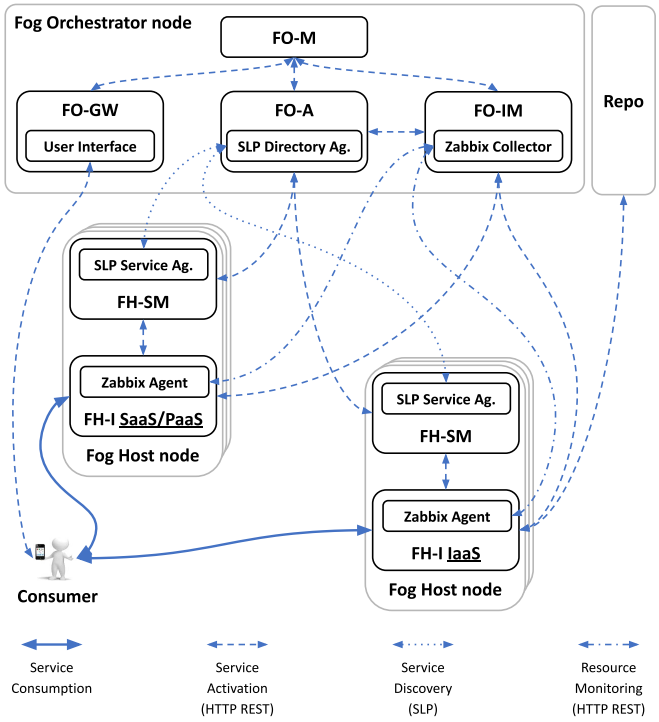


Fig. 5. The FORCH system architecture from a consumer's viewpoint, showing consumer interactions with the FO Gateway for service activation and with Fog host nodes for service consumption. The figure includes details of specific component implementation for service discovery and resource monitoring.

system, handling the interaction with service consumers and managing the activation of requested services on the underlying infrastructure. Figure 5 sketches the components of the FORCH implementation and represents a different rendition of the architecture shown in Fig. 1, useful to clarify the practical usage of the system. The figure depicts different kinds of Fog host nodes and includes details of specific component implementation for service discovery and resource monitoring. Consumer interactions with the FO Gateway for service activation and with the Fog host nodes for service consumption are also displayed.

The following subsections describe the main implementations choices we had to make to realize the proposed architecture. From a conceptual point of view, FORCH operates in a *service-centric* way, where the fundamental entity is the service [40], rather than the host node(s). The internal logic of the orchestration system associates host nodes to services, rather than the other way around, placing (the abstraction of) services at the top hierarchical level in the developed software structure. We argue that this is a sensible choice, because consumers request services, being unaware of specific host nodes, and this structure makes look-up and update operations faster and more straightforward to implement. This system design approach proved very functional for subsequent choices in code development.

A. Service discovery

After an attentive analysis of existing protocols, we implemented the mechanism of discovery of services offered by Fog host nodes by developing a general-purpose, customizable subsystem based on the Service Location Protocol (SLP), defined by the Internet Engineering Task Force (IETF) in [41]. This protocol makes use of abstractions that revolve around the concept of service as intended in the FORCH context, e.g., every service is identified by a unique string and may bear metadata that can provide additional configuration details. This solution fits well with the intended purpose of the Fog service discovery subsystem, and it allows for both active and passive search of service nodes, also including optional security features. Moreover, SLP is already widely adopted in a variety of commercial devices, guaranteeing its stability.

Our implementation of the service discovery subsystem includes a set of modules implementing the functionalities of the different SLP agents [41]. As shown in Fig. 5, an instance of the SLP Service Agent module needs to run on every node offering services, and it is operated by the FH-SM to communicate service-related information to the FO-A, where a SLP Directory Agent module collects it. In this scenario, employing a SLP Directory Agent rather than a SLP User Agent for data collection facilitates the execution of the SLP protocol with a growing number of Fog host nodes, improving scalability potential. Further details on the FORCH service discovery implementation based on SLP are available in [42].

B. Resource monitoring

In the role of monitoring system, we decided to employ Zabbix [43], a software suite providing monitoring of resources and functionality of a generic distributed system, and coming in the form of a set of agent modules and a collector module, running as background daemons. As shown in Fig. 5, the collector is employed by the FO-IM module to gather information on resource utilization of Fog host nodes, each including an instance of the agent, acting as resource monitoring apparatus of the FH Infrastructure.

C. Service requests

In the proposed implementation, reference points between consumer and orchestration level functions, and between the latter and Fog host nodes (i.e., *Fn* and *Fs* reference points in Fig. 1) are implemented as REST interfaces. Consumers only have access to a single endpoint, `/services`. Through this endpoint, they can obtain the list of available services, as well as request the activation of a service on a node of the Fog system. Data is exchanged by making use of the JSON format. The system supports multi-tenancy in a similar way to that of other orchestration platforms, where each consumer is authenticated and associated to one or more projects, and each project can have its own default configurations.

Before requesting a service activation, the consumer might retrieve the catalog of available services with a call to the method `GET /services`, causing the interaction among FORCH components as represented in Fig. 2. The response

```
Request method: GET
Request URL: http://fo-gw:6000/services
Response code: 200
Response JSON: {
  "message": "Found 2 service(s).",
  "services": [ "APP991", "FVE001" ]
}
```

Fig. 6. Example of request for available Fog services and related response including the current service catalog, performed through a REST interface implementing the reference point Fn1.

contains a list of currently available services, including all different service categories. A simple example of both request and response are shown in Fig. 6, where the current availability of two services is reported. In our implementation, the service category is implicitly determined by the format of the service identifier, e.g., APP991 and FVE001.

The consumer can then request the activation of a specific service, by means of the method `POST /services/<service_id>`, where `service_id` is the unique identifier associated to the desired service, as specified in the service catalog. As explained in Section V-B, the request triggers a sequence of interactions among the functional elements of the Fog orchestration system, which may vary depending on the availability of the service, and on whether it can be allocated or should be deployed (see Section V-A2 for the distinction). From the service consumer point of view, the interaction with the northbound interface of the Fog orchestrator level is exactly the same in both cases, regardless of the applied service model. In case of service deployment on a IaaS node, the service image can be pulled from a local or a publicly-available repository, as shown in Fig. 5.

According to a common interpretation of the REST philosophy [44], the POST method may be used to create a new resource in the system, whereas the PUT method may be used to update an existing one. In the proposed implementation, the consumer may request the activation of a service via the POST method, asking the FORCH system to activate a service without knowing whether the service is available for allocation, it needs to be deployed, or it is not available at all. The FO-M module will then perform a PUT request to the FO-IM in case the service is natively offered by an active and available Fog host node, or a POST if the service must be deployed and an eligible IaaS node is available. This distinction is transparent to the consumer, except for the difference in the status code in the response: `200 OK` if the service has been allocated, `201 Created` if the service has been deployed. Two examples of service request by a consumer and related response in case of successful allocation on an SaaS node or deployment on an IaaS node are reported in Figs. 7 and 8, respectively. In case the service activation fails, the status code would be `503 Service Unavailable`. The consumer can also invoke the DELETE method to deactivate services associated to them. For the sake of clarity, Figs. 6 to 8 only report details on the service activation process, omitting authentication information that is considered out of the scope of this discussion.

```
Request method: POST
Request URL: http://fo-gw:6000/services/APP991
Request JSON: {
  "project": "default"
}
Response code: 200
Response JSON: {
  "message": "Service APP991 active on node 10322",
  "node_ip": "192.168.64.118",
  "node_port": 8080
}
```

Fig. 7. Example of request for Fog service activation and related response notifying the successful *allocation* of the requested service on a SaaS node. The response includes the requested service endpoint.

```
Request method: POST
Request URL: http://fo-gw:6000/services/APP002
Request JSON: {
  "project": "default"
}
Response code: 201
Response JSON: {
  "message": "Service APP002 active on node 10329",
  "node_ip": "192.168.64.128",
  "node_port": 32772
}
```

Fig. 8. Example of request for Fog service activation and related response notifying the successful *deployment* of the requested service on an IaaS node. The response includes the requested service endpoint.

In our implementation, both responses of successful allocation and deployment include directly the endpoint (IP address and port) to be used for service consumption. In order to make the approach more general and robust, a symbolic name or Uniform Resource Identifier (URI) pointing at the activated service should be used, instead of a technology-specific endpoint. Additionally, the transport-layer connection between the consumer requesting the service and the orchestrator (e.g., the TCP connection in case of HTTP implementation of REST interactions) might not be able to persist for the whole time required for service activation, especially in case of service deployment on an IaaS node. For this reason, the system should respond with a `202 Accepted` code to any request, also providing a URI that the service consumer can use to poll the current status of service activation. All these improvements to the orchestrator REST interface are left for a future version of the implementation.

VII. Combinatorial Analysis of Fog Host Node Selection

After discussing the proposed Fog orchestration system architecture and mechanisms, we would like to provide a first quantitative insight on the advantages that such an orchestrator could obtain from being aware of the service model used to run the requested tasks. Therefore, in this section we give a simple quantitative assessment of the flexibility features offered by the service model-aware Fog orchestration system. The simple model we consider here is based on a combinatorial analysis of the Fog host node selection mechanism and the consequent service activation performance.

In order to make the analysis more tractable, but without loss of generality, let us make a simplifying assumption and consider only requests for APP services and two types of Fog host nodes: SaaS and IaaS. As already mentioned, a SaaS node is capable of allocating only a specific APP, whereas an IaaS node can be fully customized and can deploy any APP with a proper configuration of the underlying FVE. For the sake of this analysis, the case of PaaS nodes can be assumed similar to the SaaS nodes, if we consider that the request for a given SDP can be satisfied either by allocating a native PaaS node or by deploying the requested SDP on an IaaS node.

Let us assume there are N Fog service consumers, each requesting one out of M available APPs, $a_1, a_2, \ldots, a_M$. A given APP a_j can be randomly requested by none, one, or more than one consumer. The request set from all consumers can be represented as an array $\mathbf{r} = [r_1, r_2, \ldots, r_N]$, where the i-th request refers to the APP requested by consumer i, i.e., $r_i \in \{a_1, a_2, \ldots, a_M\}$, $\forall i = 1, 2, \ldots, N$. Due to the time needed to instantiate a virtualized appliance on an IaaS node, it is reasonable to assume that the *SaaS service activation time* $T_{S,j}$, i.e. the time required to allocate a given APP a_j on a SaaS node, is significantly smaller than the *IaaS service activation time* $T_{I,j}$, i.e. the time required to deploy an instance of the same APP a_j on an IaaS node. Therefore, $T_{S,j} < T_{I,j}$.

To be able to satisfy any possible request set $\mathbf{r} \in \{a_1, a_2, \ldots, a_M\}^N$, a suitable number K of Fog host nodes must be available. Then we have to choose how many nodes are of SaaS type (K_S) and how many are of IaaS type (K_I, such that $K = K_S + K_I$), taking into account the different service activation times needed on the two types of nodes. To make this choice, assuming that each fog host node can serve one request at a time[2] and that all requests in a given set $\mathbf{r}$ are simultaneous, we can consider three different objectives:

1) **Minimize the service activation time**. In this case, all Fog host nodes should be of SaaS type, i.e. $K_I = 0$. Since each node can allocate only a specific APP, to avoid rejecting any request due to lack of available nodes, we should have enough SaaS nodes to satisfy the worst case when all consumers request the same APP, for any APP, i.e. $K_S = MN$. The service activation time would be $T_{S,j}$ for each consumer requesting the j-th APP.
2) **Minimize the number of Fog host nodes**. In this case, all nodes should be of IaaS type, i.e. $K_S = 0$. Since each node can deploy any APP, to avoid rejecting any request we should have as many IaaS nodes as the number of consumers, i.e. $K_I = N$. The service activation time would be $T_{I,j}$ for each consumer requesting the j-th APP.
3) **Finding a trade-off between the service activation time and the number of Fog nodes**. In this case, we should find a balance between the number of SaaS and IaaS nodes, and a trade-off between the previous two objectives. To avoid rejecting any request we should have enough IaaS nodes to compensate for unavailable SaaS nodes for each APP. A possible choice would be to have as many SaaS nodes as the available APPs – one SaaS node per APP – plus as many IaaS nodes as needed in order to avoid rejecting a request in the worst case when all consumers request the same APP, i.e. $K_S = M$ and $K_I = N-1$. The average service activation time per consumer requesting the j-th APP would take a value between $T_{S,j}$ and $T_{I,j}$.

The three choices of K_S and K_I given above are to be considered as the ideal minimum numbers of Fog host nodes needed to reach the respective objective without rejecting any request. Of course, when K_S and K_I are fixed or cannot be chosen at will, it can happen that fewer nodes are available to run the requested APPs. In that case, for a given number of Fog service consumers N and available APPs M, and assuming that all requests in a given set $\mathbf{r}$ are simultaneous, we can use well-known combinatorics formulas to compute the average request rejection (or blocking) probability for any request set $\mathbf{r}$. First, we must compute the total number of possible request sets $\mathbf{r} \in \{a_1, a_2, \ldots, a_M\}^N$. Assuming that all consumers are identical, each request set is an unordered selection of N APPs identifiers. Considering that there are M APPs available and that the same APP can be requested by multiple consumers, the total number of possible request sets can be computed as the number of combinations with repetitions of N objects extracted from M classes [45]:

$$C_{\text{tot}}(N, M) = \binom{N + M - 1}{N} \tag{1}$$

where $\binom{a}{b} = \frac{a!}{b!(a-b)!}$ represents the binomial coefficient.

Then we are interested in counting the number of possible request sets $\mathbf{r}$ such that the N consumers request h distinct APPs out of M, with $1 \leq h \leq \min(N, M)$. The lower bound on h represents the case when all consumers request the same APP, whereas the upper bound is given either by the number of APPs when $N > M$, or by the number of consumers when $N \leq M$. The possible request sets for h distinct APPs are given by the number of combinations with repetitions of N objects extracted from M classes, such that at least one object is extracted from each of h distinct classes and none from the remaining $M - h$ classes. Let us choose the first h distinct objects out of M classes, recalling that there are $\binom{M}{h}$ possible ways of doing so [45]. Since the remaining $N-h$ objects must be chosen from the same h classes, we must count the number of combinations with repetitions of $N - h$ objects extracted from h classes, i.e.:

$$\binom{N-1}{N-h}$$

Therefore, the number of possible request sets such that the consumers request h distinct APPs is

$$C_h(N, M) = \binom{M}{h}\binom{N-1}{N-h} \quad h = 1, 2, \ldots, \min(N, M) \tag{2}$$

Then, assuming that any request set is generated with the same probability $p = 1/C_{\text{tot}}$, the probability that a given

[2]This is another simplifying assumption. However, the analysis can be generalized to the case where a Fog host node can simultaneously serve multiple requests by defining K in terms of the amount of resources needed to run a given set of services.

request set $\mathbf{r}$ is such that the N consumers request exactly h distinct APPs out of M is

$$P_h(N,M) = \frac{C_h(N,M)}{C_{\text{tot}}(N,M)} \quad h = 1,2,\ldots,\min(N,M) \quad (3)$$

We define the *request blocking probability* $P_B(K_S,K_I,N,M)$ as the probability that a generic request in a generic set $\mathbf{r}$ cannot be satisfied because the requested APP is not available, neither as being allocated on a SaaS node nor as being deployed on an IaaS node. Obviously, P_B depends on the choice of K_S and K_I, as well as on N and M. Let us consider the case that one SaaS node for each APP is available, i.e. $K_S = M$ and all SaaS nodes are distinct.

When $K_I = 0$, the request blocking probability is given by the probability that at least two consumers request the same APP, multiplied by the probability of being one of the consumers requesting an APP already allocated to the first consumer requesting the same APP. The event where at least two consumers request the same APP is always true when there are more consumers than APPs, so in that case we must consider all the probabilities of h distinct requests in (3) with $h \leq M$. Otherwise, when $N \leq M$ we must consider the probabilities in (3) of h distinct requests such that $h < N$, since for $h = N$ there would not be any repeated request. In all cases, the probability of having h distinct requests must be multiplied by the probability of being one of the $N-h$ consumers repeating a request for an APP already allocated by a SaaS node. Therefore, the expression of the request blocking probability for $K_S = M$ and $K_I = 0$ is given by:

$$P_B(M,0,N,M) = \sum_{h=1}^{\min(N-1,M)} P_h(N,M)\,\frac{N-h}{N}$$

When $K_I = 1$, the single IaaS node available can be used to satisfy a second request for any APP already chosen by another consumer. Therefore, in this case the request blocking probability can be computed starting from the probability that at least three consumers request the same APP. This is always true when $N > M+1$, whereas when $N \leq M+1$ it happens with all possible combinations of h distinct requests such that $h < N-1$. In all cases, the probability of having h distinct requests must be multiplied by the probability of being one of the $N-h-1$ consumers not satisfied by either a SaaS nor the IaaS node. Therefore, the expression of the request blocking probability for $K_S = M$ and $K_I = 1$ is given by:

$$P_B(M,1,N,M) = \sum_{h=1}^{\min(N-2,M)} P_h(N,M)\,\frac{N-h-1}{N}$$

The same reasoning can be extended to derive the general expression for the request blocking probability when $K_S = M$ and K_I IaaS nodes are available. Such a blocking event never happens when there is a sufficient number of IaaS nodes to satisfy any multiple requests for any APP, i.e. when $K_I \geq N-1$. Otherwise, the request blocking probability can be computed considering the probability that at least K_I+2 consumers request the same APP. This is always true when $N > M+K_I$, whereas when $N \leq M+K_I$ it happens with all possible combinations of h distinct requests such that $h < N-K_I$. In all cases, the probability of having h distinct requests must be multiplied by the probability of being one of the $N-h-K_I$ consumers not satisfied by either a SaaS nor a IaaS node, leading to:

$$P_B(M,K_I,N,M) = \begin{cases} \displaystyle\sum_{h=1}^{\min(N-K_I-1,M)} P_h(N,M)\,\frac{N-h-K_I}{N} & \text{if } K_I < N-1 \\ 0 & \text{if } K_I \geq N-1 \end{cases} \quad (4)$$

The result in (4) allows a rough dimensioning of the number of IaaS nodes that must be present in the Fog host infrastructure in order to keep the service request blocking probability under a given value. Of course the model should be generalized to the case of any number K_S of SaaS nodes. However, in that case the analysis becomes more complex because the request blocking probability depends not only on the value of K_S, but also on which APPs are supported by the SaaS nodes. Furthermore, the combinatorial analysis presented here does not take into account the temporal dynamics of service activation and consumption, as it assumes that all requests are simultaneous and must be served immediately, otherwise they are rejected. A more elaborated queuing model is needed to take into account also the time dimension, but it requires a careful estimation of the request arrival and service consumption stochastic processes, which are highly dependent on the specific types of service offered by the Fog host infrastructure. All these extensions are being considered for future developments of this work.

VIII. Validation and Performance Evaluation

The FORCH implementation described in Section VI was deployed on a testbed designed to be logically coherent with the architecture depicted in Section IV. All software components of the Fog orchestration level were implemented in Python [13], and a collection of Fog host nodes were deployed on different hardware platforms. In particular, the testbed setup included:

- a server running a Virtual Machine (VM) equipped with 4 cores and 4 GB of RAM, where all the components of the Fog orchestration level shown in Fig. 5 were hosted;
- an Intel NUC MiniPC equipped with a 4-core 8th-gen Intel i7 processor and 16 GB of RAM, used as a Fog host node;
- a RaspberryPi Single Board Computer, model 3B+, equipped with a 4-core ARMv7l processor and 1 GB of RAM, used as a second Fog host node;
- a server running up to ten VMs, each equipped with 1 core and 2 GB of RAM, used as additional Fog host nodes.

As for the Fog host node type, the NUC MiniPC was always employed as an IaaS node, running a Docker container management system as FVE and using container images stored in the public Docker repository as service images.

The RaspberryPi was configured as a PaaS node, running a general-purpose Linux-based operating system such that any relevant programming environment for software development could be installed and offered as SDP. The VMs were also running a general-purpose Linux operating system and were used alternatively as PaaS or SaaS nodes, depending on the situation, by properly installing a programming environment as SDP or a specific APP, respectively.

In order to evaluate the system performance, we generated sequences of service requests, focusing in particular on one APP (namely *Stress*, a tool that allows to generate computing resource load on a node) and one SDP (namely *Python*, an interpreter of the well-known programming language). The requests were combined in a random order, with repetitions. The specific choice of which APP/SDP to use for this purpose does not affect the service activation procedure, which is independent of the particular functionality offered by the service being activated. However, the size of the Docker image needed to deploy the service on an IaaS node could likely have an impact on the activation time, in case such image needs to be downloaded to the node from a repository. The download sizes of the two considered images were 46 MB for the Stress APP and 41 MB for the Python SDP[3].

In Section VIII-A we assess the time required for the activation of the service according to the different service models supported by FORCH. Then, in Section VIII-B we decompose the overall activation time into its main components, assessing the individual weights of the interactions among different modules that the activation process prescribes. In Section VIII-C we observe how the system handles service requests when available resources vary over time. Finally, Section VIII-D shows how the probability of the orchestration system to block the service activation request vary with the number of available Fog host nodes.

A. Service activation time

The FORCH system is able to activate the same service with different models, depending on the current availability of resources of Fog host nodes in the undelying infrastructure. Ideally, an APP or a SDP can be activated in multiple ways: by simply allocating a pre-existing instance of the APP/SDP on a SaaS/PaaS node, or by deploying (a virtualized/containerized version of) the APP/SDP on an IaaS node. From the functional point of view, these alternatives yield the same result, and consumers will not perceive any difference in the way they can access the requested service. The only perceivable discrepancy is in the activation time. For instance, to allocate an APP on a SaaS node that is running that APP already, the system simply enables the consumer to access that APP, without any other particularly complex management action needed. Therefore, we expect this procedure to be the fastest one in terms of service activation. On the other hand, in order to activate the APP on an IaaS node, the system needs to manage the deployment of a virtualized or containerized instance of the APP, depending on the specific FVE supported by the node. The time required to complete this procedure is, in principle, always larger than the activation time required in the previous case, with the possible additional burden of having to download a virtual machine or container image on the target IaaS node, if a copy of such image is not cached in the node already. This can happen in case the APP was never deployed before on that given IaaS node, or if the required image was updated or removed from the node after the last time the APP was deployed on it. Summarizing, we focus on three possible cases:

1) APP allocated on SaaS node;
2) APP deployed on IaaS node using cached image;
3) APP deployed on IaaS node after downloading the required image from the remote repository.

TABLE I
ACTIVATION TIME OF THE SAME APP ON DIFFERENT NODES

Case	Fog host node	No. of trials	Avg. activ. time (st. dev.) [s]
1	SaaS	1000	0.291 (0.177)
2	IaaS with cached image	100	1.59 (0.105)
3	IaaS without cached image	10	15.3 (0.532)

The measurements are shown in Table I, where the number of trials used to obtain the average activation time and the standard deviation are specified. The results are in line with the expectations. The quickest activation is observed when the APP is already available on a SaaS node. Allocating the APP on an IaaS node which already has the required image represents a compromise between flexibility and activation time. The slowest case is that of running the APP on an IaaS node which does not have a local copy of the required image and thus needs to download it from the repository. In that case, of course, the activation time strongly depends on where the image repository is located, the size of the image itself and the available bitrate between the Fog host node and the repository. In our testbed we used the public Docker repository, reachable from the IaaS Fog node connected to our Department's production network.

As a final note, we should also highlight that the whole testbed was hosted in our lab facilities, and therefore all communications between the Fog orchestration level components and the Fog host nodes took place in a local area network environment. While this setup could be representative of situations where the Fog orchestrator is located very close to the Fog host nodes, there could be other cases where the network delay between Fog orchestrator and Fog host nodes is not negligible. Of course, such an additional latency could have an impact on the overall activation time, which would depend on the specific deployment scenario. However, our evaluations can still be considered relevant to understand how fast the FORCH components can perform their tasks and interact with each other, as discussed in the next subsection.

[3]Both the mentioned images are publicly available on the Docker repository as `giditre/gaucho-stress:latest` and `python:slim`, respectively.

B. Service activation time breakdown

Each service activation is the result of the cooperation among the modules of the FORCH system, in a sequence of interactions that depends on the requested service and on the current resource availability. In this subsection we show the contribution of each interaction to the overall service activation time. For clock synchronization reasons, we can reliably measure these values only on a single machine at a time. Therefore, we conducted all these measurements on the same machine running the FORCH orchestration level modules.

Figures 9 and 10 show the most relevant operations that occur during the allocation of an APP on a SaaS node or its deployment on an IaaS node, respectively. The vertical length of each bar represents the duration of each interaction – or "call", as interactions among modules are realized by HTTP REST calls – involving the corresponding module as a recipient, according to the sequence diagrams in Figs. 3 and 4. The time values were measured in a single request experiment run for cases 1 and 3 in Table I, respectively. In the SaaS case, the time required to allocate the service on the host node (4) is comparable to the time required for the FO-A module to gather the list of currently-available services (3), as represented by the two rightmost bars in Fig. 9. In the IaaS case, however, the largest amount of time is required for the actual deployment (i.e., call (5) received by the FO-IM module shown in Fig. 10), which includes the execution of a new container on the chosen Fog host node. In this case, the image employed to run the container had to be pulled from the repository, therefore the amount of time required to conclude the deployment increases significantly.

It is also worth noting that in Fig. 10 the durations of the two calls received by the FO-A module – (3) to retrieve the list of all available services, (4) to retrieve the list of available FVEs – are significantly different because the short time between the two calls allows the FORCH service discovery mechanism to benefit from short-term caching of the information retrieved in call (3), resulting in a very quick response for call (4). In both Figs. 9 and 10 the vertical bar corresponding to call (1) represents the time elapsed between the consumer request and the response from the FO Gateway. Similarly, the bar corresponding to call (2) measures the time spent by the FO Mediator to complete the interactions with other components and return a response to the FO Gateway to be relayed to the consumer.

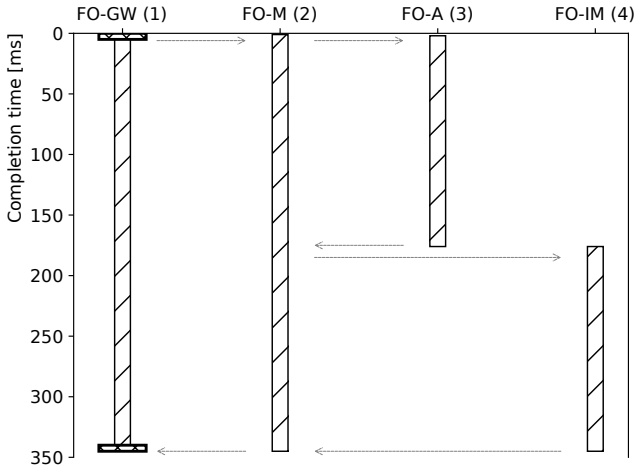


Fig. 9. Main interactions occurring during the *allocation* of an APP on a SaaS node. The names on the horizontal axis represent the entity that acts as a recipient in the specific interaction, and the numbers are to be referred to the sequence diagram in Fig. 3. The thicker markers on the FO-GW bar represent the request that the consumer sends to the system and its response, and they are not to scale with time. The arrows only provide guidance to assist in tracing this measurement back to the sequence diagram of Fig. 3, and they are not to scale with time either.

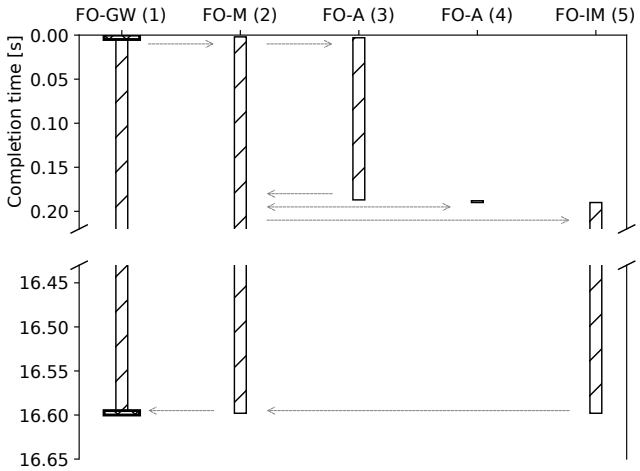


Fig. 10. Main interactions occurring during the *deployment* of an APP on an IaaS node with non-cached image. The same considerations as for Fig. 9 apply, this time referred to the sequence diagram in Fig. 4.

C. Adaptability to changing resources

FORCH bases its service activation decisions on the available amount of Fog computing resources at the time of service request, as reported by the Zabbix monitoring functions. The changing availability of host nodes in the system makes the infrastructure itself dynamic, therefore the available resources may vary over time. It is interesting to evaluate how the system behaves when receiving a constant flow of service requests while the underlying resource set changes.

For simplicity in demonstrating this feature, we ran an experiment where we considered only one resource consumption indicator, that is the CPU utilization of each Fog host node currently part of the infrastructure. The specific metric we utilized to determine the availability of a node is its 1-minute-average CPU utilization, setting the eligibility threshold to 90%. At the beginning of the experiment only an 8-CPU IaaS node was available. Then, at regular intervals of 8 minutes, a new node with 2 CPUs was activated and joined the Fog infrastructure, as a result of successful service discovery procedure. The first new node was a SaaS node offering the Stress APP, followed by a PaaS new node offering the Python SDP, then again a SaaS node followed by a PaaS node, and so on.

We launched a new service request every two minutes,

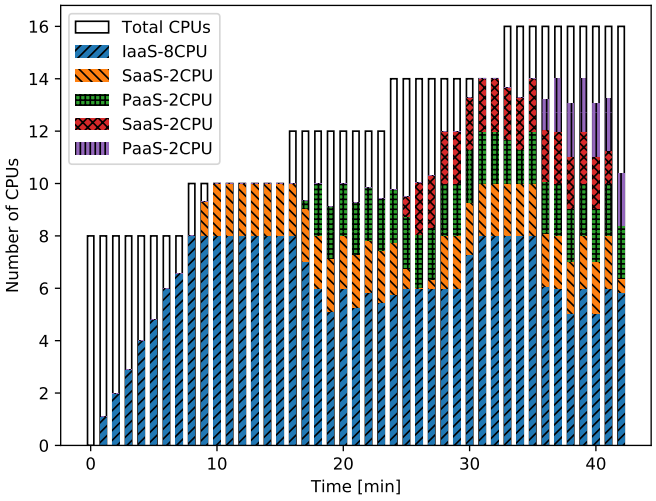


Fig. 11. CPU utilization over the whole set of Fog host nodes, during a stream of service requests. White boxes represent the total number of CPUs available at a given moment, i.e., the sum of the number of CPUs of every available Fog host node. The filling of these white boxes represents the CPU utilization on a specific Fog node.

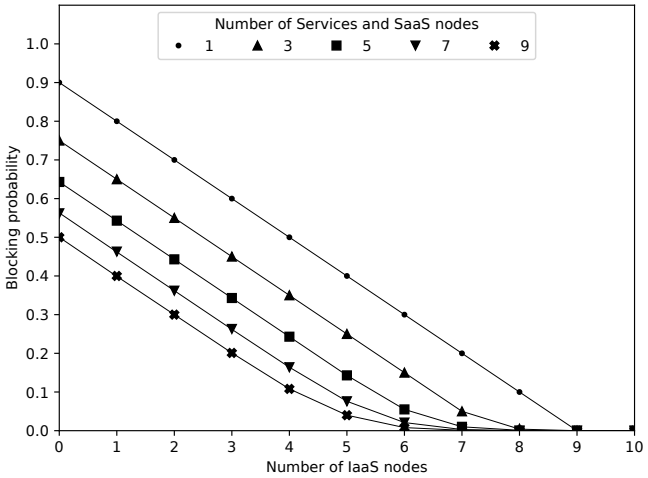


Fig. 12. Blocking probability of a service activation request as a function of the number K_I of available IaaS Fog host nodes, when $N = 10$ consumers are making simultaneous requests and for different numbers of services M. The number of available SaaS Fog host nodes is $K_S = M$. The markers indicate the values computed with the combinatorial model, while the lines represent simulation results.

starting at the beginning of the experiment. Each request could be either for the Stress APP or the Python SDP. In both cases, as soon as the service was successfully activated, the consumer kept 2 CPUs of the assigned node busy for a fixed amount of time, corresponding to 16 minutes. Both these values were chosen arbitrarily, compatibly with the duration of the experiment.

Figure 11 shows the CPU utilization of all available Fog nodes during the first 42 minutes of the experiment. The empty bars represent the total current number of CPUs, which starts at 8 and increases by 2 every 8 minutes due to new nodes joining the Fog host infrastructure. We considered here a time window where new Fog host nodes become available, making the overall pool of resources grow, to show how the system activates services in the most efficient way possible according to the current node availability. This concept would be exactly the same also if considering a shrinking resource pool. In fact, the empty bars would decrease in size over time, and services being offered by a disconnecting node would be terminated. Remedies to this are still being implemented in FORCH.

The filled bars in Fig. 11 show the current number of busy CPUs in the Fog infrastructure, with different colors and patterns representing the CPUs utilization of different nodes. At the beginning, only an IaaS node was available, so the first 4 service requests were satisfied with the deployment of the Stress APP as a Docker container. At $t = 8$ a new SaaS node joined in, and the next request was satisfied by allocating the native Stress APP on the SaaS node. Any new incoming request up to $t = 15$ was rejected due to lack of available resources. Then a new PaaS node joined in, and a new request for the Python SDP was satisfied. Meanwhile, some of the APPs deployed on the IaaS node terminated, so the corresponding CPUs became available again and could be used to satisfy new requests, while two more nodes joined in at $t = 24$ and $t = 32$.

The experiment demonstrated that FORCH is capable of selecting the most convenient service model to satisfy a new request based on the current resource availability, by prioritizing SaaS/PaaS nodes when possible. This is shown by the fact that CPUs of SaaS/PaaS nodes are always filled first when new request arrive (e.g., at $t = 25$ and $t = 27$), and FORCH resorts to using IaaS nodes only when no SaaS or PaaS node offers the requested APP/SDP (e.g., at $t = 30$).

D. Request blocking

As explained in Section V-B, each service activation request faces the possibility of being rejected (i.e., blocked), due to the lack of available resources in the Fog infrastructure. The probability of this happening is modelled in Section VII. In order to validate the model, we developed a simulator that allows to compute the blocking probability as a function of the number of consumers, services, SaaS and IaaS Fog host nodes, using a Monte Carlo approach. A simulation-based evaluation allows to perform a sufficient number of iterations (10^5 in our case) for statistical accuracy within a reasonable amount of time.

Figure 12 shows that, for $K_S = M$, the request blocking probability computed with the combinatorial model matches the simulated values, confirming that, when there is one SaaS host node per service, the model is valid. The curves show how the blocking probability decreases as the number K_I of IaaS nodes increases, until it reaches zero for $K_I = N - 1$, for different numbers of services and SaaS nodes M, and for $N = 10$ consumers. As expected, the higher the number of services (and of SaaS nodes offering them) the lower the blocking probability for a sub-optimal number of IaaS nodes, as it is more likely that the consumers will pick a service that has an available SaaS node offering it. This simple analysis can be used to make a rough estimation of the number of IaaS

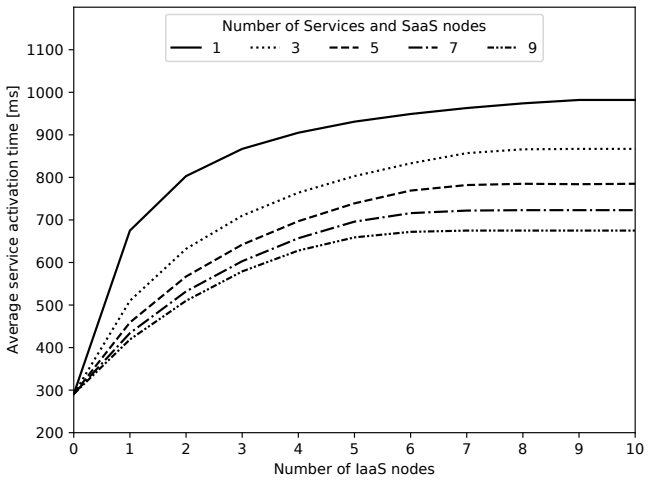


Fig. 13. Average service activation time as a function of the number K_I of available IaaS Fog host nodes, when $N = 10$ consumers are making simultaneous requests and for different numbers of services M. The number of available SaaS Fog host nodes is $K_S = M$.

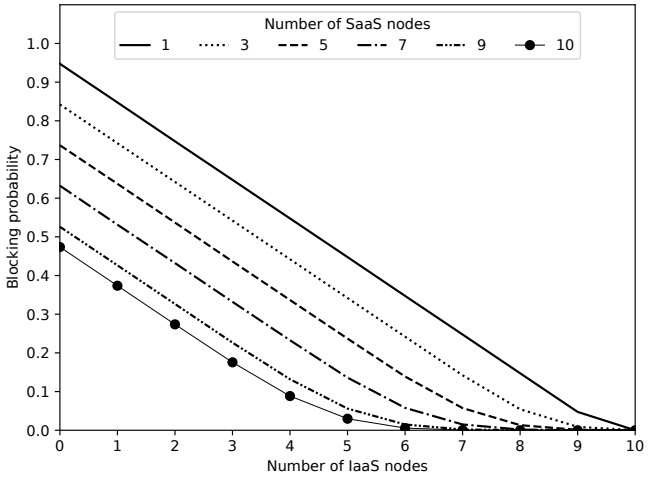


Fig. 14. Blocking probability of a service activation request as a function of the number K_I of available IaaS Fog host nodes, when $N = 10$ consumers are making simultaneous requests, with $M = 10$ services and different numbers of available SaaS Fog host nodes K_S. The dots represent the values computed with the combinatorial model, which is valid for $K_S = M$.

nodes that should be put in place to reach a certain level of performance.

Figure 13 shows the average service activation time obtained by simulation under the same conditions as in Fig. 12. The computation is based assuming that each single service allocated on a SaaS node or deployed on an IaaS node takes the average time measured in Section VIII-A. The results confirm the intuitive trade-off that for a higher number of IaaS nodes, thus for a lower request blocking probability, the average service activation time gets higher due to more deployments than allocations, stabilizing on a value that decreases as the number of available SaaS nodes increases.

The simulation results presented in Fig. 14 show how the blocking probability behaves when the number of services differs from the number of available SaaS nodes, with the former being fixed to $M = 10$. In this case the simulation results deviate from the model, which is valid only when $K_S = 10$.

IX. Conclusion

The adoption of different service models can be satisfactorily extended to the Fog Computing scenario, allowing for great flexibility in deploying services in an environment where conditions are inherently different from those found in a conventional Cloud context. In this paper we proposed an architecture for service model-aware orchestration in Fog Computing scenarios, where the available resources may vary over time, and demonstrated the feasibility of the approach by presenting an original working implementation of the proposed architecture, along with relevant experimental validation. We discussed the most important design choices and included an accurate description of components and operations of the proposed orchestration system, including insights on how it is able to handle critical functions such as service discovery and resource monitoring. As a result of our investigation, we showed that Fog services can be effectively deployed in a matter of a few seconds, or even in less than one second when suitable Fog nodes are available, taking advantage of the awareness of the available service models. We also showed that this outcome can be achieved on top of a dynamic infrastructure built with consumer-grade hardware. We included a combinatorial analysis of the service model-aware resource selection process, to obtain a first-approximation quantitative assessment of the advantages of the proposed approach in terms of service availability.

Future research directions include a more general formalization of the analytical model to compute the request blocking probability, so as to relax some of the simplifying assumptions and take into account also the time dimension of service activation and consumption. An advanced study on smart or optimized approaches to determine the best node to activate a service on, in case more than one of them is eligible, represents another potential research topic. From an implementation viewpoint, future work contemplates the addition of support for composite services and for other flavors of the XaaS model (including the highly promising Function-as-a-Service approach), as well as the refinement of some internal mechanisms of the current implementation, in the direction of a more robust and reliable orchestration system.

Acknowledgment

This work was partially funded by the Italian Ministry of Education, University and Research (MIUR) though a grant awarded to the Department of Electrical, Electronic and Information Engineering “Guglielmo Marconi” (DEI) of the University of Bologna, as part of the *Departments of Excellence* initiative aimed at funding the best Departments of Italian State Universities. The work was also partially funded by the University of Bologna as part of the AlmaIdea project PRONE - *Programmable Networks for Emergency Trials*.

REFERENCES

[1] Y. Ku, D. Lin, C. Lee, P. Hsieh, H. Wei, C. Chou, and A. Pang, "5G radio access network design with the fog paradigm: Confluence of communications and computing," *IEEE Commun. Mag.*, vol. 55, no. 4, pp. 46–52, Apr. 2017.

[2] M. Chiang and T. Zhang, "Fog and IoT: An overview of research opportunities," *IEEE Internet Things J.*, vol. 3, no. 6, pp. 854–864, Dec. 2016.

[3] S. Miano, F. Risso, M. V. Bernal, M. Bertrone, and Y. Lu, "A framework for eBPF-based network functions in an era of microservices," *IEEE Trans. Netw. Service Manag.*, vol. 18, no. 1, pp. 133–151, Mar. 2021.

[4] M. Iorga, L. Feldman, R. Barton, M. J. Martin, N. S. Goren, and C. Mahmoudi, "Fog computing conceptual model," The National Institute of Standards and Technology, SP 500-325, Mar. 2018. [Online]. Available: https://doi.org/10.6028/NIST.SP.500-325

[5] C. Mouradian, D. Naboulsi, S. Yangui, R. H. Glitho, M. J. Morrow, and P. A. Polakos, "A comprehensive survey on fog computing: State-of-the-art and research challenges," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 1, pp. 416–464, First Quarter 2018.

[6] M. Mukherjee, L. Shu, and D. Wang, "Survey of fog computing: Fundamental, network applications, and research challenges," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 3, pp. 1826–1857, Third Quarter 2018.

[7] A. Mseddi, W. Jaafar, H. Elbiaze, and W. Ajib, "Intelligent resource allocation in dynamic fog computing environments," in *2019 IEEE 8th International Conference on Cloud Networking (CloudNet)*, Coimbra, Portugal, Nov. 2019, pp. 1–7.

[8] D.-N. Vu, N.-N. Dao, W. Na, and S. Cho, "Dynamic resource orchestration for service capability maximization in fog-enabled connected vehicle networks," *IEEE Trans. on Cloud Comput.*, Early access, 2020.

[9] A. Bozorgchenani, D. Tarchi, and G. E. Corazza, "Centralized and distributed architectures for energy and delay efficient fog network based edge computing services," *IEEE Trans. Green Commun. and Netw.*, vol. 3, no. 1, Mar. 2019.

[10] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," The National Institute of Standards and Technology, SP 800-145, Sep. 2011. [Online]. Available: https://doi.org/10.6028/NIST.SP.800-145

[11] V. Cardellini, G. Mencagli, D. Talia, and M. Torquati, "New landscapes of the data stream processing in the era of Fog computing," *Future Generation Computer Systems*, vol. 99, pp. 646–650, 2019.

[12] G. Davoli, D. Borsatti, D. Tarchi, and W. Cerroni, "FORCH: An orchestrator for fog computing service deployment," in *2020 IFIP Networking Conference (Demo Session)*, Jun. 2020, pp. 677–678.

[13] Unibo gaucho: Forch. [Online]. Available: https://github.com/giditre/unibo_gaucho

[14] *Multi-access Edge Computing (MEC); Framework and Reference Architecture*, ETSI GS MEC 003, Rev. 2.2.1, Dec. 2020. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/MEC/001_099/003/02.02.01_60/gs_mec003v020201p.pdf

[15] D. Borsatti, G. Davoli, W. Cerroni, and C. Raffaelli, "Enabling industrial IoT as a service with multi-access edge computing," *IEEE Communications Magazine*, vol. 59, no. 8, pp. 21–27, Aug. 2021.

[16] *IEEE Standard for Adoption of OpenFog Reference Architecture for Fog Computing*, IEEE Std. 1934-2018, Jun. 2018.

[17] M. Antonini, M. Vecchio, and F. Antonelli, "Fog computing architectures: A reference for practitioners," *IEEE Internet Things Mag.*, vol. 2, no. 3, pp. 19–25, Sep. 2019.

[18] M. Aazam, S. Zeadally, and K. A. Harras, "Fog computing architecture, evaluation, and future research directions," *IEEE Commun. Mag.*, vol. 56, no. 5, pp. 46–52, May 2018.

[19] Z. Wen, R. Yang, P. Garraghan, T. Lin, J. Xu, and M. Rovatsos, "Fog orchestration for internet of things services," *IEEE Internet Comput.*, vol. 21, no. 2, pp. 16–24, Mar. 2017.

[20] P. Pop, B. Zarrin, M. Barzegaran, S. Schulte, S. Punnekkat, J. Ruh, and W. Steiner, "The FORA fog computing platform for industrial IoT," *Information Systems*, vol. 98, p. 101727, May 2021.

[21] Y. Jiang, Z. Huang, and D. H. K. Tsang, "Challenges and solutions in fog computing orchestration," *IEEE Netw.*, vol. 32, no. 3, pp. 122–129, May 2018.

[22] P. Habibi, M. Farhoudi, S. Kazemian, S. Khorsandi, and A. Leon-Garcia, "Fog computing: a comprehensive architectural survey," *IEEE Access*, vol. 8, pp. 69 105–69 133, 2020.

[23] T. Goethals, F. De Turck, and B. Volckaert, "Live demonstration of a highly scalable fog service orchestrator," in *2021 IEEE International Conference on Network Softwarization (NetSoft 2021) (Demo Session)*. IEEE, 2021.

[24] R. Vilalta, V. López, A. Giorgetti, S. Peng, V. Orsini, L. Velasco, R. Serral-Gracia, D. Morris, S. De Fina, F. Cugini, P. Castoldi, A. Mayoral, R. Casellas, R. Martinez, C. Verikoukis, and R. Munoz, "TelcoFog: A unified flexible fog and cloud computing architecture for 5G networks," *IEEE Commun. Mag.*, vol. 55, no. 8, pp. 36–43, Aug. 2017.

[25] S. Tuli, R. Mahmud, S. Tuli, and R. Buyya, "FogBus: A blockchain-based lightweight framework for edge and fog computing," *Journal of Systems and Software*, vol. 154, pp. 22–36, Aug. 2019.

[26] P. Habibi, S. Baharlooei, M. Farhoudi, S. Kazemian, and S. Khorsandi, "Virtualized SDN-based end-to-end reference architecture for fog networking," in *2018 32nd International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, Krakow, Poland, May 2018, pp. 61–66.

[27] F. Faticanti, F. De Pellegrini, D. Siracusa, D. Santoro, and S. Cretti, "Throughput-aware partitioning and placement of applications in fog computing," *IEEE Transactions on Network and Service Management*, vol. 17, no. 4, pp. 2436–2450, 2020.

[28] N. Morkevicius, A. Venčkauskas, N. Šatkauskas, and J. Toldinas, "Method for dynamic service orchestration in fog computing," *Electronics*, vol. 10, no. 15, p. 1796, 2021.

[29] M. Tortonesi, M. Govoni, A. Morelli, G. Riberto, C. Stefanelli, and N. Suri, "Taming the IoT data deluge: An innovative information-centric service model for fog computing applications," *Future Generation Computer Systems*, vol. 93, pp. 888–902, Apr. 2019.

[30] S. Hoque, M. S. De Brito, A. Willner, O. Keil, and T. Magedanz, "Towards container orchestration in fog computing infrastructures," in *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2. IEEE, 2017, pp. 294–299.

[31] S. Tuli, S. Poojara, S. N. Srirama, G. Casale, and N. Jennings, "Cosco: Container orchestration using co-simulation and gradient based optimization for fog computing environments," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[32] F. Faticanti, L. Maggi, F. De Pellegrini, D. Santoro, and D. Siracusa, "Fog orchestration meets proactive caching," in *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, 2021, pp. 878–883.

[33] C. Chang, S. Narayana Srirama, and R. Buyya, "Indie Fog: An efficient fog-computing infrastructure for the Internet of Things," *Computer*, vol. 50, no. 9, pp. 92–98, Sep. 2017.

[34] A. Bozorgchenani, D. Tarchi, and W. Cerroni, "On-demand service deployment strategies for Fog-as-a-Service scenarios," *IEEE Commun. Lett.*, vol. 25, no. 5, pp. 1500–1504, May 2021.

[35] R. Moallemi, A. Bozorgchenani, and D. Tarchi, "An evolutionary-based algorithm for smart-living applications placement in fog networks," in *2019 IEEE Globecom Workshops (GC Wkshps)*, Waikoloa, HI, USA, Dec. 2019.

[36] D. Tarchi, S. Grandi, and W. Cerroni, "Android-based implementation of a fog computing and networking environment," in *2019 IEEE Wireless Communications and Networking Conference (WCNC)*, Marrakesh, Marocco, Apr. 2019.

[37] D. Milojicic, "The edge-to-cloud continuum," *Computer*, vol. 53, no. 11, pp. 16–25, Nov. 2020.

[38] X. Wei, C. Tang, J. Fan, and S. Subramaniam, "Joint optimization of energy consumption and delay in cloud-to-thing continuum," *IEEE Internet Things J.*, vol. 6, no. 2, pp. 2325–2337, Apr. 2019.

[39] F. van Lingen, M. Yannuzzi, A. Jain, R. Irons-Mclean, O. Lluch, D. Carrera, J. L. Perez, A. Gutierrez, D. Montero, J. Marti, R. Maso, and J. P. Rodriguez, "The unavoidable convergence of NFV, 5G, and fog: A model-driven approach to bridge cloud and edge," *IEEE Communications Magazine*, vol. 55, no. 8, pp. 28–35, 2017.

[40] J. Kempf, C. E. Perkins, and E. Guttman, "Service Templates and Service: Schemes," RFC 2609, Jun. 1999. [Online]. Available: https://rfc-editor.org/rfc/rfc2609.txt

[41] M. D. Day, C. E. Perkins, J. Veizades, and E. Guttman, "Service Location Protocol, Version 2," RFC 2608, Jun. 1999. [Online]. Available: https://rfc-editor.org/rfc/rfc2608.txt

[42] M. Valieri, "Dynamic resource and service discovery in fog computing," Master's thesis, University of Bologna, Bologna, Italy, 2021. [Online]. Available: http://amslaurea.unibo.it/22265/

[43] Zabbix - Enterprise-class Open-source Distributed Monitoring Solution. [Online]. Available: https://www.zabbix.com/

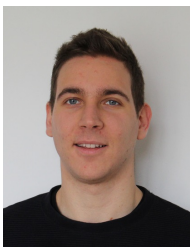
[44] R. T. Fielding and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content," RFC 7231, Jun. 2014. [Online]. Available: https://rfc-editor.org/rfc/rfc7231.txt

[45] R. A. Brualdi, *Introductory Combinatorics*, 5th ed. Pearson, 2010.

Gianluca Davoli (Member, IEEE) received his M.Sc. and Ph.D degrees in Telecommunications Engineering from the University of Bologna, Italy, in 2017 and 2021, respectively. He is currently a Post-Doc Researcher and Adjunct Professor at the same institution. His research interests revolve around communication networks, focusing on the new approaches to programmability, management, and monitoring of software-based network infrastructures.

Walter Cerroni [M'01, SM'16] is an Associate Professor of Communication Networks at the University of Bologna, Cesena Campus, Italy. He graduated in Telecommunication Engineering from the University of Bologna in 1999, and obtained his Ph.D. in Electronic and Computer Engineering from the same institution in 2003. In 1999 he was an intern at the Alcatel Corporate Research Center, Dallas, TX, USA, partly with a grant as Visitor Researcher from the University of Texas at Dallas. From 2003 to 2005 he was a Research Associate at the National Inter-University Consortium for Telecommunications (CNIT), Italy. From 2005 to 2020 he was an Assistant Professor at the University of Bologna. In 2008 he was Visiting Assistant Professor at the School of Information Sciences, University of Pittsburgh, USA. Walter Cerroni coauthored more than 140 articles published in the most renowned international journals, magazines, and conference proceedings. His recent research interests focus on multiple aspects of control, management and orchestration of communication network infrastructures, including software-defined networking, network function virtualization, multi-access edge computing, fog computing, service function chaining, intent-based networking systems. He serves/served as Series Editor for the IEEE Communications Magazine, Associate Editor for the IEEE Communications Letters, and Technical Program Co-Chair for IEEE-sponsored international workshops and conferences.

Davide Borsatti received his B.S. and M.S. in Telecommunications Engineering from University of Bologna in 2016 and 2018, respectively. He is currently enrolled in the Electronics, Telecommunications, and Information Technologies Engineering PhD program from the University of Bologna. His research interests include NFV, SDN, Intent Based Networking, MEC and 5G Network slicing.



Mario Valieri received his Bachelor's in Electronic Engineering and Master's Degree in Telecommunications Engineering from the University of Bologna, Italy, in 2018 and 2021 respectively. He is currently a Firmware Engineer at Eggtronic, Modena, Italy. His interests focus on embedded devices, digital hardware design and computer science.

Daniele Tarchi (S'99–M'05–SM'12) was born in Florence, Italy in 1975. He received his M.Sc. degree in Telecommunications Engineering and Ph.D. degree in Informatics and Telecommunications Engineering from the University of Florence, Florence, Italy, in 2000 and 2004, respectively. From 2004 to 2010, he was a Research Associate with University of Florence, Italy. From 2010 to 2019 he was an Assistant professor at the University of Bologna, Bologna, Italy. Since 2019 he has been an Associate Professor at the University of Bologna, Italy. He is the author of more than 130 published articles in international journals and conference proceedings. His research interests are mainly on Wireless Communications and Networks, Satellite Communications and Networks, Edge Computing, Fog Computing, Smart Cities, and Optimization Techniques. He has been involved in several national and international research projects, and collaborates with several foreign research institutes. Prof. Tarchi is an IEEE Senior Member since 2012. He is Editorial Board member for IEEE Wireless Communications Letters, IEEE Transactions on Vehicular Technology and IET Communications. He has been symposium co-chair for IEEE WCNC 2011, IEEE Globecom 2014, IEEE Globecom 2018 and IEEE ICC 2020, and a workshop co-chair at IEEE ICC 2015.

Carla Raffaelli is Associate Professor at the University of Bologna. She received her M.Sc. and Ph.D degrees in electronic and computer engineering (University of Bologna, Italy), in 1985 and 1990, respectively. Her research interests include performance analysis of telecommunication networks, switch architectures, optical networks and 5G networks. She actively participated in many National and International research projects, such as the EU funded ACTS-KEOPS, the IST-DAVID, the e-photon/One and BONE networks of excellence. She is author or co-author of more than 150 conference and journal papers mainly in the field of optical networking and network performance evaluation. She regularly acts as a reviewer for top international conferences and journals and serves as Technical Program Committee member in several IEEE International Conferences, like ICC, Globecom and HPSR. Since October 2013 she is a member of the editorial board of the journal Photonic Network Communications by Springer. She is associate editor of IEEE OJ- COMS since its foundation. She is IEEE Senior Member since 2016 and OSA member. She is the Director of the International Telecommunications Engineering Master's Degree at the University of Bologna, Italy.