Dynamic On-Demand Fog Formation Offering On-the-Fly IoT Service Deployment

Hani Sami and Azzam Mourad

***Abstract*—With the increasing number of IoT devices, fog computing has emerged, providing processing resources at the edge for the tremendous amount of sensed data and IoT computation. The advantage of the fog gets eliminated if it is not present near IoT devices. Fogs nowadays are pre-configured in specific locations with pre-defined services, which limit their diverse availabilities and dynamic service update. In this paper, we address the aforementioned problem by benefiting from the containerization and micro-service technologies to build our on-demand fog framework with the help of the volunteering devices. Our approach overcomes the current limitations by providing available fog devices with the ability to have services deployed on the fly. Volunteering devices form a resource capacity for building the fog computing infrastructure. Moreover, our framework leverages intelligent container placement scheme that produces efficient volunteers' selection and distribution of services. An Evolutionary Memetic Algorithm (MA) is elaborated to solve our multi-objective container placement optimization problem. Real life and simulated experiments demonstrate various improvements over existing approaches interpreted by the relevance and efficiency of (1) forming volunteering fog devices near users with maximum time availability and shortest distance, and (2) deploying services on the fly on selected fogs with improved QoS.**

***Index Terms*—IoT, fog computing, on-demand fog formation, edge computing, docker, Kubernetes, Kubeadm, container placement, evolutionary memetic algorithm, micro-services.**

I. Introduction

IN TODAY'S fast growth and high intelligence, we are encountering a vast increase in the number of IoT devices changing the way we live. This leads to a humongous volume of data that has to be dealt with before going to the cloud with the help of fog nodes located at the edge next to IoT devices [1]. The purpose of having fog nodes located next to users can be summarized as a computation resource for filtering the data, processing power for the data before it goes to the cloud, minimizing the workload on the cloud, achieving faster response time, and diminish their energy consumption by reducing data transmission over the network. Fog is a perfect solution for resource constrained devices and users in need of a service running nearby to get a better Quality of Service (QoS). However, current work in the literature are considering fog devices pre-configured in specific locations next to a group of known users and running the same known services all the time. This limits the fog advantages in terms of having them available all the time next to the user in need and does not allow the fog to change and update services in its hosting environment on-demand. Accordingly, there is a need for having an architecture that can help in creating fogs on-demand, and can adapt or configure the installed services that should be updated, removed, or changed dynamically.

In parallel, the rise of the containerization technology is opening the door for interesting solutions serving the fog computing objectives. Containers are services running on a device by using their actual operating system to provide them with services. This makes containers more lightweight and gives them an advantage over virtual machines which use a full copy of the operating system and are much heavier on devices [2]. It is easier now to just have an abstracted operating system with all the environments needed to run multiple services in multiple containers that are a copy of images pulled from image repositories like Docker Hub. Docker and Kubernetes are the main containerizations and orchestration technologies used nowadays [3]. Moreover, the fast emergence of micro-service architecture, where services are designed to be decoupled and lightweight, makes them perfect candidates to be deployed and executed on containers. In this paper, we benefit from the containerization and micro-service technologies to address the aforementioned problems of the existing statically formed fog computing architectures and solutions in the literature.

In this context, we propose a dynamic on-demand fog computing framework based on Kubeadm and Docker with the presence of volunteering devices. Images embedding lightweight micro-services can be deployed and run efficiently on the fly even on devices with limited resources [4]. The advantage of supporting on the fly deployment technique is to push only current needed services which were not predicted to be requested. Moreover, the motivation behind using volunteering devices to join the fog network is to increase the available resources capacity wherever possible, which leads to maintained services availability everywhere. Our proposed on-demand creation serves as a solution to overcome the limitation of fog availability, usage of fogs on statically defined locations, hosting pre-configured devices, and embedding pre-selected services. A master-worker nodes architecture is implemented with the help of a Kubernetes Utility called Kubeadm [5], which uses docker to monitor the status of

Manuscript received May 23, 2019; revised September 11, 2019 and December 3, 2019; accepted December 6, 2019. Date of publication January 1, 2020; date of current version June 10, 2020. This work was supported by the Lebanese American University. The associate editor coordinating the review of this article and approving it for publication was J. Sa Silva. *(Corresponding author: Azzam Mourad.)*

The authors are with the Department of Computer Science and Mathematics, Lebanese American University, Beirut 10017, Lebanon (e-mail: azzam.mourad@lau.edu.lb).

Digital Object Identifier 10.1109/TNSM.2019.2963643

1932-4537 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Bibliothèque ÉTS. Downloaded on June 25,2021 at 03:51:42 UTC from IEEE Xplore. Restrictions apply.

containers running on every worker node through the master node. Moreover, another problem arises to select the best volunteers that can host the different required services. In this regard, we formulate this problem as a multi-objective container placement optimization problem and provide an Evolutionary Memetic Algorithm to solve it [6]. Using heuristics, our decision model efficiently dictates for each service on which volunteer to be pushed.

To overview our approach and illustrate its contributions and advantages, we take three real life scenarios that require dynamic creation of a nearby fog: (1) An IoT device consuming significant energy while sending lots of data all the way to the cloud, (2) a user frequently requesting a particular service from the cloud and requiring better QoS, and (3) a crime taking place in an area where an image processing service need to be pushed on volunteering fog(s) to process the image and identify faces in the scene. In this context, our framework based on some criteria allows the cloud to decide on the need for fog serving near this device. We have three possible actions that the cloud can take. First, a Kubeadm cluster is already created in the area where we need a serving fog. In this case, the cloud forwards the pushing requests to master node of the ready orchestrator and let it deploy the service on the available volunteer. Second and third, in case a kubeadm cluster is not ready, two actions can take place based on the status (e.g., battery level), the behavior (e.g., number or type of requests) of the user sending the data, and the available volunteering resources in this area. The cloud can either decide to directly join a fog to its cluster and push images of services to it or to initialize an orchestrator first and then let it do the job of joining a fog to its new cluster. If it is either an orchestrator or a fog, the container placement algorithm running on the orchestrator outputs the near optimal distribution of services on a group of available volunteers based on the proximity and available resources. Many conflicting objectives are taken into consideration such as the number of services pushed to volunteers, quality of service, distance from volunteers to users, and time availability.

With the help of volunteering devices, containerization, orchestration, and micro-service technologies, we are able in this paper to overcome the aforementioned limitations and achieve the following contributions:

1) Proposing a novel on-demand fog computing architecture embedding dynamic creation and management of volunteering devices and efficient on the fly deployment of IoT services.
2) Providing generic and adaptable multi-objective optimization model to formulate container placement problem and adapt its evaluation to different contexts such as the available resources, priority of required services, proximity from IoT devices, and time availability.
3) Offering efficient orchestration approach in a dynamic fog environment.

The road-map for this paper is as follows. A discussion of some important background information and related work is presented in Section II. We illustrate our problem in Section III. In Section IV, we propose our dynamic on-demand fog computing architecture. We define and formulate the multi-objective optimization model and prove its complexity in Section V. In Section VI, we choose the evolutionary memetic algorithm to solve our problem. Implementation and experimental results are depicted in Section VII to evaluate our approach and prove the efficiency of our container placement model against competing architectures and placement solutions. Finally, in Section VIII, we conclude our work and open the door for future research directions.

II. Background and Related Work

We present in this section some background information and relevant state of the artwork done in the literature about the use of volunteering resources as infrastructure and containerization technology in the context of fog computing. We also provide the related work we counted on to build our multi-objective optimization model and other competing solutions.

A. *Fog Computing Trends and Requirements*

In the surveys [7] and [8], authors provide detailed descriptions on existing architectures to serve the fog computing paradigm and highlight on their main limitations. The challenges of managing heterogeneous fogs resources and defining the proper strategy of deploying a fog environment in real time are identified as the ongoing trends. Authors also discussed the importance of the fog presence near IoT devices to support real time-sensitive applications such as the real-time fire detection to dispatch fleets of robots to the proper destination at the proper time, and the virtual reality applications that require fogs presence. Therefore, on the fly fog initialization is required to support such applications. In this paper, we focus on providing an architecture that supports fog management and services deployment on available volunteering heterogeneous fog resources using micro-services, containerization, and orchestration technologies. As stated in [7], micro-services pushed using containerization technology is a promising solution to provide smooth deployment of cloud services near the edge because of the agility and independent distribution of micro-services and lightweight nature of containers installment.

B. *Fog Containerization*

Before containers, services were pushed to fog devices using virtual machines; some approaches are still using them [9]. Containers are proven to be more lightweight than virtual machines as they have many advantages [2]. Therefore, recent literature work is shifting to the use of containerization technology for service deployment. The potential of using Docker technology to run containers on fog devices with the ability to adjust services hosted whenever needed is proven by authors in [10]. A model was proposed by authors in [11] where dynamic deployment of services on helper nodes of the main server using Docker is possible. So it is feasible to remove, add, stop, and run any service on a physically known fog anytime. The main limitation of this paper is that the fogs along with their locations are already known and may not be near users. In this paper, we call such devices **static fogs**.

Authorized licensed use limited to: Bibliothèque ÉTS. Downloaded on June 25,2021 at 03:51:42 UTC from IEEE Xplore. Restrictions apply.

Furthermore, in their work, the user is requesting to push services to fogs, and it is not on-demand decided by the server.

In the recent work done by [12], authors focused on the ability to use lightweight Docker containerization technology to support service provisioning over IoT devices. Their main contribution is to show how lightweight containerization technology can manage IoT resources by hosting services on them.

The authors in [13] and [14] tried to build a framework that can dynamically push services to fogs present in an area where these devices are known for the orchestrator (static fogs).

C. Containers Placement Problem

López-Pires and Barán [15] proposed an interactive memetic algorithm to solve the proposed multi-objective formulation of the virtual machine placement problem. The goal is to find an efficient distribution of VMs on corresponding available hosts with respect to the conflicting objective functions. This problem can be mapped to our container placement problem by considering the VMs as containers, and the available hosts to run the VMs as the volunteering fog devices.

The purpose in [16] is to provision fog resources and provide the efficient distribution of services on fogs. They mathematically formulated the problem and took into account the number of pushed services and proximity from the user; however, they did not consider the time availability of this fog to serve. Our approach counts on volunteering devices, so this objective affects the optimal decision. This is proven in our experiments below. Moreover, they used genetic algorithm to solve their problem; in our case, we are using the evolutionary memetic algorithm to get a good solution at early stages and to minimize the chances of halting in a local optimal compared to the genetic algorithms. This is achieved using the probabilistic bit string mutation that maximizes the number of explored possible solutions. We also use a probabilistic local search that is proved to find feasible solutions at early stages by improving the current solutions in the population and provide a balance exploitation of the different objective functions [15].

D. Volunteering Fog as Infrastructure

Following the history of the growth of cloud computing, several approaches tried to augment the constrained devices' performance by offloading their heavy work to remote server or to the cloud [17], [18]. The main limitations of such approach are the network congestion and delays caused by long communications with the cloud. After modernizing the computation power of mobile devices, it becomes more efficient to rely on these devices to host services, perform computation, or even building a secure mobile environment [19]. In the same context, the work of authors in [20], [21] showed the ability and benefit of bringing new volunteering fog devices to help the main one in case they are next to each other, as well as the ability of distributing the load of requests on many fog volunteers. A model for dynamic resources allocation was proposed by [21] to assign requests to fogs based on their advertised resources in real time. Although the fogs are accepting requests related to some particular services in this paper, they didn't discuss how these services are running on these devices. An elaboration on the ability to use vehicular nodes as infrastructure to build the fog environment was proven by authors in [22]. They proved the ability to use moving and parked vehicles as infrastructure to support moving users using Vehicular Fog Computing.

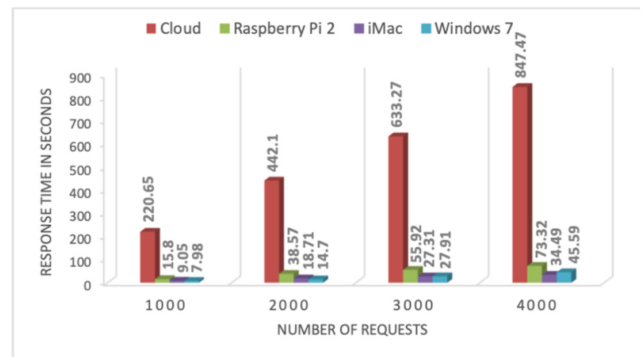

Fig. 1. Response Time of Requests Sent From User to the Cloud Compared to the Fog on Different Types of Devices.

E. Analysis

Based on the above discussed limitations, there is a need for fog presence on-the fly and on-demand. To the best of our knowledge, none of the literature work has proposed a solution serving the fog computing paradigm to create fogs on the fly on any type of devices. In addition, no one proposed a solution for the container placement problem. Our proposed approach uses Memetic to generate distributions of containers on selected fog volunteers using heuristics based on specific attributes in a way that suits the user's needs, minimizes the communication overhead and resources consumption, and maximizes the number of pushed services efficiently.

III. Problem Illustration

Fog devices are not available everywhere, and some IoT users are not covered by them. Whenever a fog is not available to serve a user in need, the requests of this user have to go to the cloud. This causes a delay in network communication. Consequently, the user is not experiencing the needed quality of service, and some others are wasting energy waiting for their requests to be served. To shed light on the importance of the presence of fog devices when needed, we considered the traditional fog case where a user is requesting a service from it against requesting it from the cloud. For this purpose, different types of devices were used to run the same service as the cloud and placed two hops away from the user. The used service replies to users with an HTML page using the Flask framework [23]. The recorded response time of sending a group of requests simultaneously to the cloud and various devices is illustrated in Fig. 1. As shown in the graph, even constrained devices have better performance than the cloud. This can be interpreted by the networking delays for the requests to reach the cloud and go back to the corresponding user. The response time of sending 1000 requests simultaneously to the cloud is 214 seconds, while it takes the Raspberry Pi2 (Pi2) device 7

Authorized licensed use limited to: Bibliothèque ÉTS. Downloaded on June 25,2021 at 03:51:42 UTC from IEEE Xplore. Restrictions apply.

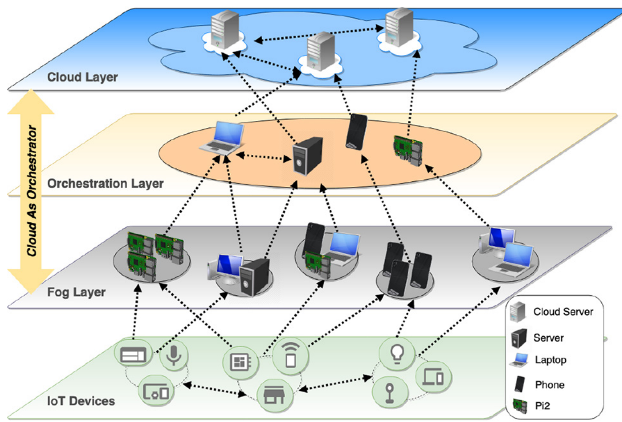

Fig. 2. Proposed Framework - High-Level Architecture.

seconds to reply to all the requests. This experiment explores the importance of fog presence nearby and also illustrates the losses in response time and network delay in case of their absence. In this context, our proposed approach constitutes a solution for forming on-demand fog with on the fly service deployment for serving the need of nearby IoT devices.

IV. Proposed Architecture

The presence of fog devices enables services to reside at the edge next to IoT devices. This implies less transmission power from sensors, therefore less power consumption on the IoT device. Moreover, the vast amount of data generated by IoT devices requires data processing with high capacity and low delay. This requires powerful processing to serve requesting users. Therefore, we design our proposed framework with the purpose of supporting IoT devices when needed anytime by deploying/managing container-enabled services after forming ready fog clusters near them on the fly.

A high level overview of the architecture can be found in our previous work [24]. In this section, we discuss the architecture of our proposed framework, illustrated in Fig. 2. The on-demand fog Kubeadm cluster is built using the volunteering resources and cloud only. The Kubeadm utility is used because it supports Docker and allows us to create custom Kubernetes clusters using any constrained devices. The architecture is composed of four layers: cloud, orchestration, fogs, and IoT/users.

The cloud is responsible for taking a decision on-demand where we need serving fog(s) for a particular service based on users' behaviors and recurrent requests coming from a particular location. It is also responsible for adding orchestrators or fogs in the areas where we have a lack of serving nodes. We have three scenarios that can take place to let the cloud perform different actions accordingly:

1) In case a Kubeadm cluster using volunteering devices is already built in the target area, then the initialization time of a master node and joining time of the volunteering node is saved. This way, the orchestrator of the cluster can directly deploy services to available volunteers.

TABLE I
Node Architecture Per Layer

Layer	Component
Cloud	- Kubeadm Required Modules - Kubeadm Environment Initializer - Orchestrators Manager - Container Placement - Orchestrator - Volunteers Manager
Orchestration	- Kubeadm Required Modules - Fogs Manager - Container Placement - Fogs - Profiler
Fog	- Kubeadm Required Modules - Fog Client - Profiler

2) In case there were no volunteering devices available to prepare the Kubeadm cluster, then we have two possible scenarios.
   a) If there are no available resources to initialize a new orchestrator near users, the cloud let its volunteering devices join its cluster to serve as fog nodes.
   b) If enough resources are available in the target location, the cloud asks a suitable volunteer to initialize a Kubeadm cluster (to become a master node), which will ask the volunteers to join as fogs.

The orchestrator or the master node of a Kubeadm cluster runs the services for deciding on adding and removing specific fogs in its cluster as well as deploying, updating, or removing services from them. For example, services that are not in use for a certain period should be removed by the orchestrator.

Fog is the worker nodes of a Kubeadm cluster that is either created by the cloud master or another on-demand initialized orchestrator near it. This fog is a volunteering device which asked to join by sending requests to the cloud. In this context, the cloud or orchestrator(s) chooses from the available list of volunteers the best set of devices to host a specific service on these fogs while taking into consideration their profiles and based on the decision of the optimization model. Until a fog is available to serve a user, the requests continue their flow normally to the cloud until the newly created fog's IP is published to the users.

An IoT device or a user sends requests normally to the cloud if there is no fog in the area, and receives a fog IP address once any is available next to it. This way, the user is experiencing a better quality of service (QoS) by having a faster response time, less communication overhead, and a full device dedicated to serving one or a group of users/IoT devices.

A description at each level of the components embedded in each node of the architecture is presented in Table I. All nodes have to run the Kubeadm containerization required modules. A decision module for selecting volunteering nodes as orchestrators or fogs has to run on the cloud and the orchestrator. Finally, the profiler components should be running on the orchestrator and fog volunteers.

In the sequel, we present a description of the modules running on each layer of the architecture. Each of these modules is implemented as python scripts using the Flask Web services

Authorized licensed use limited to: Bibliothèque ÉTS. Downloaded on June 25,2021 at 03:51:42 UTC from IEEE Xplore. Restrictions apply.

framework and pushed to the Docker Hub repository after building their Docker files. These images are used later by any device joining a cluster.

A. Kubeadm Required Modules

Kubeadm helps in building a best practice of the Kubernetes cluster in a very secure, easy, and extensible way [5]. When using Kubeadm, the device can range from a raspberry pi to a server. Docker should always be running on all devices. Kubelet component starts running on each end. It is responsible for keeping the communication alive with the master. It also checks the health of services running and the status of the device as well. Kubectl is the command line tool which should be installed to control Kubeadm cluster. A Container Network Interface solution (CNI) should be installed in the cluster and on each device to let the pods communicate with each other. An example of a CNI network is Flannel [25]. Kubeadm will not run if any of these dependencies are missing. The master and worker nodes should have the Kubeadm containerization required modules installed. When the master is initialized, it will have a unique hashed token to be sent to devices to join the cluster as worker nodes. When the worker node is ready, different images are pulled and run inside its pods once joined.

B. Kubeadm Environment Initializer

The cloud starts initializing the first orchestrator in a location when it receives a certain number of requests above a certain threshold from this location. For this paper, we are assuming that there is always a list of available suitable volunteers to use. The following are the provided functionalities:

1-Taking decisions for selecting the most suited volunteering orchestrators: The Decision module for selecting volunteering nodes as orchestrator discussed in Section V is triggered. It takes as input the profiles of available volunteers in a particular location. Its output is the most suitable volunteer to be assigned as a new orchestrator.

2-Joining volunteers to orchestrator: Whenever the volunteering orchestrator of a location is ready, the cloud prompts all remaining volunteering devices in this location to join the running Kubeadm cluster monitored by the newly created orchestrator. Therefore, the Kubeadm environment is ready to handle pushing services without any initialization delays needed directly.

3-Guaranteeing highly available cluster: This module checks the time remaining and the availability of all the orchestrators running. It also informs the orchestrator selection algorithm about the need of having a new orchestrator before the old one terminates. If resources are available, always another master node should be created next to the initial one to achieve a highly available cluster (HA: highly available, this feature is supported by Kubeadm where both master nodes join the same network [5]). Also, if the initial orchestrator goes down, the secondary one will be replacing it. This way, the framework avoids delays in creating a new master node. The initial master node always asks for a secondary one from the cloud. A limitation in Kubeadm is that whenever the master node goes down, the whole cluster gets down as well [5].

C. Orchestrators Manager

In the following, we describe the functionalities offered by the orchestrator manager which is a case of an orchestrator of orchestrators running on the cloud:

1-Collecting Users Requests: Users requests are collected from the server logs and can be used for further analysis. These logs represent historical data of users behavior of a particular location.

2-Taking decision of Orchestrator/Fog Creation: Based on users behavior, number of requests coming, and the level of urgency of services needed near them (e.g., a user can send urgent request when they are losing a lot of energy, or they need a better QoS), the cloud decides if either an orchestrator should be created to initiate the fog node through, or it is a time-sensitive situation where a fog should be created directly. Unless a Kubeadm cluster is available in the target location, we cannot ignore the initialization time of a cluster needed in real life simulations. In addition, if a decision is taken to push services to a device, it assigns priority levels to each image/service to be pushed before others. Priorities are assigned because available resources to host containers of services might not be enough to host all of them. On the other hand, the cloud might also decide that there is no need to initialize a device and host any services on it at this moment. It is a complex decision to be made, and it might require a machine learning model that can classify the user behavior and requests coming into three categories as follows: no need for a fog, an orchestrator should be created, or a fog should be created, and a priority level for each service if needed. However, the implementation of this unit is out of the scope of this paper. For now, we are assuming in our experiments that the decision of creating an orchestrator or a fog is already done by the decision module. Moreover, our core approach is to let the cloud just decide on a fog needed where the cloud is its orchestrator; however, we extended the capabilities of this module with an approach for having the orchestrator initialized near fog devices and not always running on the cloud. The advantages of this extension are:

- Reducing the delay in choosing and creating new fog devices to join the cluster in a highly dynamic environment of fogs coming in and out. In other words, the delay of the communications between the cloud and fogs near edge devices is eliminated.
- Distributing the load coming to the cloud on orchestrators, especially when fogs joining the cluster have short time availability, which renders the situation very dynamic.
- Avoiding to have a single point of failure where the cloud is the only orchestrator for a huge number of fog devices.

3-Joining and Pushing Services to Volunteering Orchestrators: If the cloud decides that a fog needed in an area where an orchestrator exists, it sends this orchestrator the list of needed services to be pushed. If a volunteer is ready, it pushes the required and needed services to it. If the orchestrator is not available and the user can tolerate some delay, the cloud calls the decision module for selecting orchestrators to initialize the best suited one, and then push

Authorized licensed use limited to: Bibliothèque ÉTS. Downloaded on June 25,2021 at 03:51:42 UTC from IEEE Xplore. Restrictions apply.

the services required by our framework to be running on the master node. If it is a time-sensitive situation, the cloud calls the decision module to select volunteering fog(s) for joining (Container Placement). When the fog is ready, images will be downloaded on it.

D. Fog Manager

In this section, we discuss the functionalities of the fog manager, a case where we have an orchestrator of fogs. The orchestrator can be on the cloud or on volunteering devices near fogs. When the service needed is urgent, the cloud directly pushes join commands to the volunteering device(s) to join its cluster as a volunteer. The volunteer is considered as a fog device when it hosts at least one service. By letting the volunteer directly join the cloud cluster as fog, we are saving the delay of creating a new Kubeadm cluster. The following are the provided functionalities:

1-Getting List of Volunteers and Services: The fog manager accepts requests coming from the cloud containing a list of services to be pushed and another one of available volunteers. When the cloud is the orchestrator of fogs, it reads the volunteers records available from its database.

2-Joining and Pushing Services to Fogs: The fog manager assigns services to fogs if already running and suitable for the new services. If there is no fog available, it adds new ones to the cluster.

3-Calling Decision Module to Select Volunteers: The orchestrator triggers the decision module for selecting volunteering fog nodes and assigning the proper services to them based on the available capacities of volunteers, proximity from users, and time availability with respect to the requested resources by the services and their priorities (Section V).

4-Balancing Load: This component monitors the number of requests served by each fog and decides on a Load Balancer if needed, which means we need more containers to serve this service. In this case, the load of one fog is distributed on others in peak time only, or new containers are created on the same fog if possible. This feature is provided by Kubernetes [5].

5-Backing-Up Fogs: Similar to how Back-Ups of orchestrators are created, this module checks the remaining time of a fog inside the cluster and tries to instantiate a new one before the old one leaves the cluster.

6-Monitoring Volunteer Status and Services: This is done by checking the profile and services running on each fog. If any service is not being requested by users, it will be removed. By default, Kubelet automatically restarts any service that fails and report the failure to the orchestrator [5]. If the failure occurs more than a provided threshold, the orchestrator in our framework initializes another one with the required services running before excluding it from the cluster.

7-Publishing New Fogs IPs: The orchestrator informs users of the new fog's IP address by getting their list from the cloud.

E. Container Placement - Decision Module for Selecting Volunteer Nodes as Orchestrator or Fog and Service Placement

This module is triggered by the orchestrator or fog manager where an orchestrator or a fog is setup based on its decision. The efficient selection among nodes is based on several criteria and profiling measures, which is considered a multi-objective optimization problem. A full description of the problem and an implementation of the algorithm to solve it are provided in Section V (formulated as container placement problem). If an orchestrator should be selected and initialized to cover a particular area, this algorithm will have the responsibility to send the master's initialization commands to the selected orchestrator. If a fog is to be chosen, a kubeadm joining command will be sent to the volunteer to join the cloud's cluster, and suitable services will be pushed to selected volunteers based on the decision of the optimization model.

F. Volunteers Management

The cloud accepts volunteering requests from users and stores them in a database. The data contains the profile of the user and most importantly, its location. This database is accessed by the elaborated volunteer management algorithm whenever the cloud wants to choose an orchestrator or fog.

G. Profiler

This module allows the master node of a cluster to request all the necessary information about the device including the computation power, number of CPUs, current CPU usage, size of the memory, memory usage, size of the disk, disk usage, battery level if available, name of the device, and more importantly its location and time availability. The profile is updated frequently and requested when needed from the cloud or the orchestrator.

H. Fog Client

In this section, we discuss the fog client functionalities running on the volunteering fogs or any potential volunteer.

1- Sending Requests to Join a Cluster: When a volunteering device wants to advertise its resources to join as orchestrator or cloud, the fog client sends join requests containing its profile to the cloud to keep a record of it.

2- Keeping Cloud Updated: The volunteer device replies to the cloud by sending its updated version of the profile once requested based on timestamps.

V. Container Placement Problem

This section is divided into two parts. In the first one, we define our multi-objective optimization problem and prove its complexity. In the second part, we mathematically formulate our problem by providing the input and output data, the constraints that should be applied when building a solution, and finally, the objective functions.

A. Problem Definition

Given two sets of services as Docker containers $\mathrm{S} = \{\mathrm{s1}, \mathrm{s2}, \ldots, \mathrm{sn}\}$ to be pushed to a certain location, and available volunteering devices $\mathrm{D} = \{\mathrm{d1}, \mathrm{d2}, \mathrm{d3}, \ldots, \mathrm{dm}\}$ in the same location, we have to find the best distribution of these services on such devices while taking into consideration their different demands in terms of the resources consumption and priority factor for each service, whereas the devices are

Authorized licensed use limited to: Bibliothèque ÉTS. Downloaded on June 25,2021 at 03:51:42 UTC from IEEE Xplore. Restrictions apply.

providing their available resources in terms of capacity, time availability, and location with respect to the requesting group of users. Selecting the best set of devices and optimally finding distribution of services on them is complex and considered as NP-hard problem.

Theorem 1: Multi-objective Optimization Problem is NP-Hard.

The well known Bin-Packing problem is NP-Hard [26]. Therefore, by reducing our problem to the Bin-Packing problem, we can prove that our problem is NP-Hard. The traditional bin packing problem is described as follows. Suppose we have a set of objects with different volumes that you need to pack inside a finite number of bins of some capacity or volume. The aim is to try to maximize the total object packed in the bin, and to minimize the number of used bins. Our problem can be mapped to the Bin-Packing problem as follows. Each bin is our available volunteering device that has some resources capacities, and the objects are the service images we need to assign for these hosts. Our objective is to maximize the number of pushed service while minimizing the number of active hosts, in addition to other objective functions. Thus our problem is NP-hard.

B. Problem Formulation

We aim to optimize the number of pushed services to devices, the number of active devices to host all of these services, the QoS, the survivability factor, and the distance range from the hosting device to the requesting node.

1) Input Data: The proposed formulation of the container placement problem models a set of services S to be pushed to a particular location from the cloud to a set of available hosts H in this location.

The set of services is represented as a matrix $S \in \mathbb{R}^{n*4}$ corresponding to four input features. Each service S requires processing resources of CPU, memory, disk space, and provides a priority level as follows:

$$S_i = \left[S_{cpu_i}, S_{m_i}, S_{d_i}, S_{p_i}\right],\ \forall i \in 1, \ldots, n,$$

where:

$$\begin{aligned}
&S_{cpu_i} : Processing\ requirements\ of\ S_i \\
&S_{m_i} : Memory\ requirements\ of\ S_i \\
&S_{d_i} : Disk\ requirements\ of\ S_i \\
&S_{p_i} : Priority\ level\ of\ S_i\ with\ respect\ to\ other\ services \\
&n : Number\ of\ services\ to\ be\ pushed.
\end{aligned}$$

The set of available hosts are represented as a matrix $H \in \mathbb{R}^{n*5}$ corresponding to 5 input features. Each host is offering the available resources in terms of CPU, memory and disk space, in addition to the time availability (Survivability Factor), and its current location with respect to the group of users. The volunteering host can be mathemically represented as:

$$H_j = \left[H_{cpu_j}, H_{m_j}, H_{d_j}, H_{su_j}, H_{l_j}\right],\ \forall j \in 1, \ldots, m,$$

where:

$$\begin{aligned}
&H_{cpu_j} : CPU\ availability\ on\ host\ H_j. \\
&H_{m_j} : Memory\ availability\ on\ host\ H_j. \\
&H_{d_j} : Disk\ space\ availability\ on\ host\ H_j. \\
&H_{su_j} : Time\ where\ H_j\ is\ available\ to\ host\ services\ . \\
&H_{l_j} : Location\ of\ H_j\ with\ respect\ to\ targeted\ group\ of \\
&\quad\quad users. \\
&m : Number\ of\ available\ hosts.
\end{aligned}$$

2) Output Data: A calculated solution K should indicate a complete placement of each Service S_i into H_j, considering the applied multi-objective optimization criteria. A placement is represented as a matrix $K = K_{ij}$ of dimensions $(n * m)$ where $K_{ij} \in 0, 1$ indicates if S_i is located or not to be pushed on host H_j. In addition, arrays L and R are derived from the output matrix K. L represents the set of active hosts that at least host one service by setting L_j to 1, otherwise it is 0. On the other hand, R represents the set of services that are pushed by setting R_i to 1 and to 0 otherwise. L and R are described as follows:

$$\begin{aligned}
L_j &= \left(\sum_{i=1}^{n} K_{ij} > 0\right) \quad ? \quad 1 \quad : \quad 0 \\
R_i &= \left(\sum_{j=1}^{m} K_{ij} > 0\right) \quad ? \quad 1 \quad : \quad 0.
\end{aligned} \tag{1}$$

3) Constraints (Physical Resources Capacity of H_j): A service S_i can be hosted on H_j if the host capacity will not reach its maximum after allocating the service to it. The capacity is formulated as the CPU, Memory, and Disk space available to be used by our framework on the host, and the service requirement as the CPU, Memory, and Disk space usages. This constraint can be expressed in mathematical format as follows:

$$\sum_{i=1}^{n} S_{cpu_i} \times K_{ij} \leq H_{cpu_j} \tag{2}$$

$$\sum_{i=1}^{n} S_{m_i} \times K_{ij} \leq H_{m_j} \tag{3}$$

$$\sum_{i=1}^{n} S_{d_i} \times K_{ij} \leq H_{d_j} \tag{4}$$

$\forall j \in 1, \ldots, m$, i.e., for all available hosts H_j

Minimum Survivability Time: To avoid a very high dynamic environments of hosts joining and dropping from the cluster, we specify a minimum time availability offered by any device to be able to join the cluster as follows:

$$\forall j \in 1, \ldots, m \quad H_{su_j} \geq D \quad where \quad D \in \mathbb{N}. \tag{5}$$

Unique Placement of S_i: In the same targeted location, a service S_i should not be pushed more than once to an available host H_j unless it is a special where a load balancer is needed. The above is expressed as follows:

$$\sum_{j=1}^{m} K_{ij} \leq 1,\ \forall i \in 1, \ldots, n \tag{6}$$

Authorized licensed use limited to: Bibliothèque ÉTS. Downloaded on June 25,2021 at 03:51:42 UTC from IEEE Xplore. Restrictions apply.

Applying Priorities to Services: Each service has priority level from 1 to t. t is the highest level of priority, which means that this service should be pushed first. If the total available resources capacity in a location is not enough to host all services, the following equation should be applied:

$$\sum_{j=1}^{m} K_{ij} = 1,\ \forall i \ such \ that \ S_{p_i} = t. \tag{7}$$

Weights Summation: We proposed the weight multiplication with each objective function in order to provide more flexibility in prioritizing an objective function over others. This weight is a decimal value between 0 and 1 where the sum of all weights is equal to 1. This approach is also known as the method of adjustable weights [27].

For example, given W1 = 0.3 and W2 = 0.1, the W1 value will affect the maximization results of the sum of objective functions, therefore, f1 value will have more effect on the overall optimization decision. It is expressed as follows:

$$W_{f1} + W_{f2} + W_{f3} + W_{f4} + W_{f5} = 1. \tag{8}$$

4) Objective Functions (Number of Pushed Services Maximization): The aim is to Maximize the number of pushed services to devices in the chosen population.

$$F1 = max\left(\left(\sum_{i=1}^{n}\sum_{j=1}^{m} K_{ij}\right) \times W_{f1}\right) \tag{9}$$

where W_{f1} is the weight of the objective function maximizing the pushed services. By maximizing the number of pushed services, we guarantee that all users' requests for services are satisfied and deployed next to them.

QoS Maximization: The quality of service is improved by ensuring that the maximum number of services with high priorities are pushed. This objective is proposed in [28] as follows:

$$F2 = max\left(\left(\sum_{i=1}^{n} C^{P_i} \times P_i \times R_i\right) \times W_{f2}\right) \tag{10}$$

where:

C^{P_i}: is a large constant that prioritize services with high P_i over others with low value of P. P is a score indicating the priority level of a service.

W_{f2}: is the weight of the objective function maximizing QoS.

Survivability Factor Maximization: Maximizing the time a device is available to host particular services is expressed as:

$$F3 = max\left(\left(\sum_{j=1}^{m} H_{su_j} \times L_j\right) \times W_{f3}\right) \tag{11}$$

where W_{f3} is the weight of the objective function maximizing survivability factor. By maximizing the time a device is available to host a service, we are neglecting the time of initializing a new hosting device to get ready for serving users.

Host Distance Minimization: Minimizing distance to the user requesting the service is expressed as:

$$F4 = min\left(\left(\sum_{j=1}^{m} H_{l_j} \times L_j\right) \times W_{f_4}\right) \tag{12}$$

where W_{f_4} is the weight of the objective function minimizing the host distance. By minimizing the distance, we guarantee the minimum networking delay, which means getting faster response time, in addition of minimizing the energy losses on the user side.

Active Hosts Minimization: The aim of this objective function is to minimize the number of active hosts to save initialization time of devices when joining the cluster, using less energy with less hosts running, and a better way for the orchestrator to monitor less devices. It is expressed as:

$$F5 = min\left(\left(\sum_{j=1}^{m} L_j\right) \times W_{f_5}\right) \tag{13}$$

where W_{f_5} is the weight of the objective function minimizing active hosts.

After having the optimization functions declared, the multi-objective optimization problem becomes:

$$y = f(x) = [f_1(x), f_2(x), f_3(x), f_4(x), f_5(x)] \tag{14}$$

where:

$$\begin{aligned} f_1(x) &= \text{number of pushed services;} \\ f_2(x) &= \text{QoS;} \\ f_3(x) &= \text{survivability factor;} \\ f_4(x) &= \text{host distance;} \\ f_5(x) &= \text{number of active hosts;} \end{aligned} \tag{15}$$

subject to:

$$\begin{aligned} &e_1(x) : \text{processing resource capacity of } H_s; \\ &e_2(x) : \text{memory resource capacity of } H_s; \\ &e_3(x) : \text{minimum survivability;} \\ &e_4(x) : \text{unique placement of } S; \\ &e_5(x) : \text{services with high priority first;} \\ &e_6(x) : \text{sum of all objective function weights} = 1; \end{aligned} \tag{16}$$

VI. Memetic Algorithm for Container Placement Problem

It is important to get the pareto set of solutions for the container placement problem in a short period. The memetic algorithm which is built on top of the genetic algorithm is a suitable solution for such problems. It is not only characterized by an evolutionary optimization strategy, but also by a local optimization (local search) algorithm that can guarantee near optimal solutions in early generations [6]. The memetic algorithm proposed in [15] and adapted to solve our optimization problem is illustrated in Algorithm 1. Every chromosome in our proposed MA solution is represented by an $i \times j$ matrix. In each chromosome K, allele K_{ij} denotes a decision for service i placement on host j, where $K_{ij} \in [0, 1]$.

Authorized licensed use limited to: Bibliothèque ÉTS. Downloaded on June 25,2021 at 03:51:42 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1 Multi-Objective Memetic Algorithm

Data: Set of containers
Result: Pareto set approximation P_{known}
1: Check if the problem has a solution
2: Initialize set of solutions P_0
3: P_0' = repair infeasible solutions of P_0
4: P_0'' = apply local search to solutions of P_0'
5: Update set of non-dominated solutions P_{known} from P_0'
6: $t = 0$
7: $P_t = P_0''$
8: **While** (stopping criterion is not met), do
9: Q_t = selection of solutions from $P_t \cup p_{known}$
10: Q_t' = crossover and mutation of solutions of Q_t
11: Q_t" = repair infeasible solutions of Q_t'
12: Q_t"' = apply local search to solutions of Q_t"
13: increment t
14: Update set of non-dominated solutions P_{known} from Q_t"'
15: P_t = fitness selection from $P_t \cup Q_t$"'
16: **End while**
17: **Return** Pareto set approximation p_{known}

First, we check that the problem has at least a feasible solution where containers of services can be hosted on available volunteers before moving to step 2. Next, we initialize a random set of solutions P_0 by randomly assigning images of services to available volunteers. In step 3, we look at the set of available solutions in P_0, and repair any violation of our constraints (e.g., services to be hosted on a device with resource consumption greater than available capacity of the volunteer). This violation are repaired in three ways and illustrated in Algorithm 2: (1) Moving containers to other available volunteering devices, (2) adding unused volunteers to the list of running one and move Docker containers to them, and (3) removing the container from the list of services to be pushed. Step 4 of the Memetic Algorithm (MA) is to apply a probabilistic local search method presented in Algorithm 3 to optimize feasible solutions. If the probability is less than 0.5, we maximize the number of pushed services in their order of priority following line 4. This will lead to maximizing $f_1(x)$, $f_2(x)$ and $f_3(x)$. On the other hand, if we have probability >0.5, we minimize the number of available volunteers which minimizes $f_4(x)$ and $f_5(x)$ (less number of hosts will lead minimization in the total distance). Then the Pareto set approximation is generated at step 5. After the initialization of step 6, normal selection, crossover, and mutation operators are applied, infeasible solutions are repaired, optimization of solutions is done using probabilistic local search, iteration counter is incremented, and finally, the Pareto set is updated if any improvements happened. After that, a new population is selected. This process keeps on repeating until the algorithm reaches several iterations. Finally, the fittest set of the solution p_{known} is returned. In this MA, we use the binary tournament for selecting individuals from the population to apply crossover and mutation on them. The crossover operator used is the single point cross-cut, where selected individuals belonging to the ascending population are replaced with descending ones. This crossover approach is discussed in [29]. In this work, we make use of the bit string mutation. In this operator, every gene is mutated with probability $1/n$

Algorithm 2 Infeasible Solution Reparation

Data: Infeasible Solution
Result: Feasible Solution
1: $\mathit{feasible} = \mathit{false}$; $i = 1$
2: **While** $i \leq n$ *and* $\mathit{feasible} = \mathit{false}$ **do**
3: **if** *it is possible* **then**
4: move S_i to H_j' $(j \neq j')$
5: **else**
6: **if** S_j does not have priority level
7: Remove S_i from list of services to be pushed
8: **else**
9: Moving S_i to other available volunteers H_j in P_{known}
10: **end**
11: **end**
12: **end while**
13: **return** Feasible Solution

Algorithm 3 Probabilistic Local Search

Data: Set of Feasible Solutions P_t'
Result: Set of Feasible optimized Solutions P_t"
1: Probability: Random value between zero and one
2: **While** there are solutions not verified **do**
3: **if** Probability < 0.5 **then**
4: We remove containers placed on H_j and run them on H_j' if resources available are enough, and then assign any unselected service on H_j if resources are available after sorting them with priority level
5: **else**
6: We assign all services S_i needed to available H_j
7: devices depending on resources requirement, and
8: then we discard all H_j and assign all services S_i
9: to new set of volunteers H_j' that can host them
10: **end**
11: **end while**
12: **return** Set of Optimized Solutions P_t"

where n is the number of services. This approach guarantees a uniform opportunity of mutation over individuals with a small probability, maximize diversity, improves the search space, and prevents the stagnancy in a local optimum.

The complexity of this algorithm is divided into four parts as discussed in [30]. Let M and N be the number of chromosomes and number of nodes respectively. The generation of M chromosomes, the crossover, the mutation, and the local search complexity time. The MA starts off using $O(M \times (n-1) \times log(n-1))$ time units to generate the random population. Also let p_m and p_c be the probability of the mutation and crossover respectively. The number of offsprings generated by the crossover uses $O(N \times p_c \times [M \times (N+1)])$, while the ones created by the mutation consumes $O(p_m \times [M \times (N+1)])$ of time units. The local search algorithm consumes $O(n)$. Therefore the combined time complexity of the MA is:

$$O((M \times (n-1) \times log(n-1)) + (N \times p_c \times [M \times (N+1)]) + (p_m \times [M \times (N+1)] + N)).$$

VII. Experiments and Simulation Results

To demonstrate the relevance and efficiency of our approach, we performed Three main experiments and simulations covering different aspects and real-life scenarios. The first experiment

Authorized licensed use limited to: Bibliothèque ÉTS. Downloaded on June 25,2021 at 03:51:42 UTC from IEEE Xplore. Restrictions apply.

aims to show the importance of our proposed on-demand fog formation approach where services are hosted on the fly in an already created Kubeadm cluster next to requesting users. We compared our results to existing architectures, including hosting the service on the cloud, using VMs to push the service (like most of the current approaches in the literature), using static fogs, and placing the fog far from the user.

In some real-life scenarios, resources may not be available to meet the assumed requirements of the first experiment. In this regard, we provide the second experiment tailored to show how our framework behave in such cases. Although these cases rarely occur, it is worth studying their effect. In the aforementioned two experiments, we are assuming optimal selection of volunteering fogs and distribution of services on them, which is not the case in real life environment. Some volunteering devices may have shortage on resources. Selecting them as orchestrators and/or fogs may affect negatively the service behavior and consequently our approach. In this regard, we prove through simulation in the third experiment, that the solution of the container placement optimization problem generates efficient results in favor of our proposed framework while comparing it to existing work. We also prove that our approach is able to scale and converge to a near optimal solution when the input is large.

In the context of the first two experiments, we built a suitable environment composed of many nodes and a typical network topology where a user can access services running on machines present in a lab linked using Ethernet and others on a cloud server. We set up the environment on a Linux EC2 T2.small instance running on Amazon Web Services cloud (AWS). We used in the lab an HP Core i7 laptop running Windows7 with 8GB of RAM, an iMac corei5 with 8 RAM, and a Pi2 to demonstrate the usage of mobile resource constrained device. The user requesting the service is two hops away from the lab. We installed Kubeadm containerization required tools on all devices. The service used for testing in this environment is composed of a simple image representing a micro-service. The container of the image receives requests from users and replies back with an HTML file. The purpose of choosing this service is to check the networking delay to receive responses from the hosting fog rather than counting on the computation power of them for now. The service is implemented in Python using Flask framework [23] along with its Docker file that is pushed to the Docker Hub repository for further usage. For the last experiment, Containernet simulator is used. Containernet is a fork of the known Mininet simulator. Containernet provides the flexibility of creating a whole network on a single machine. This simulator lets us create hosts that support the use of Docker images. We shifted from the real-life experiments to the simulated Containernet because we can easily adjust the distance between hosts by adding link delay between them. The time availability can also be implemented by breaking the link from a user to a fog while the simulation is running.

A. *Experiment 1 - On-The-Fly Service Deployment on Volunteering Fog Compared to Existing Approaches*

In this experiment, we prove the efficiency and feasibility of our on-demand fog scheme when having the Kubeadm orchestrator and cluster of volunteers ready beforehand. This case is mostly faced in real life environment. The time of deploying the image (service) on the fly on a volunteering fog is considered. The improvement achieved by our approach is compared to existing work including the use of static fogs [11], [13], [14], use of VMs to host services on fogs [9], and possibility of initializing the fog far from the user. This latter case is possible when resources are available on static fogs placed far from the user. All of these approaches are also compared to the standard use of the cloud to serve IoT devices.

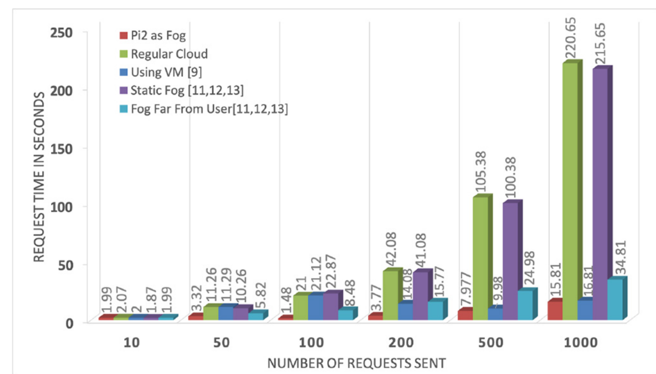

Fig. 3. Response Time of Simultaneous Requests Sent to On-Demand Fog Running On Pi2 Against Existing Approaches.

1) Experiment Setup: The cluster and the fog were previously initialized using the Kubeadm commands, and the Pi2 is the target fog on which the service needed by the user should be deployed. The request is sent to the orchestrator to push the service for use. We downloaded a minimal Ubuntu VM image and compared its download time to Ubuntu Docker image. The Internet speed was set to a limit of 50mbps. To simulate a fog that is far from the user, we added networking delay of 80ms on a certain port of the machine.

2) Experimental Results: Fig. 3 shows the difference between the response time of a different number of requests sent to the cloud against the ones sent to the on-demand nearby, static, and far fogs, including the time of pushing the service from the concerned repository either as VM or a container. The results are promising where major time saving is achieved in the case of on-demand deployment on volunteering fogs near the user with respect to accessing the service in the other scenarios. The result of each scenario can be compared to our on demand volunteering approach as follows.

- Static fogs can be placed far from the user, causing a networking delay between the user and serving fog. As shown in Fig. 3, the response time when requesting from the static fog is double the response time of our approach. For example, at 1000 requests, the response time in case of the static fog is 34s compared to 16.81s when placing the fog on a volunteer near the user.
- In case of using VMs to download the service on the machine, it takes around 80s to download the VM at a speed of 50m/s compared to 5s in case of downloading the same service in a form of a container having a size of 28MB. After 500 requests, the VM is ready and starts serving with low response time. In case the service needs to be updated or a new service is requested, the user has

Authorized licensed use limited to: Bibliothèque ÉTS. Downloaded on June 25,2021 at 03:51:42 UTC from IEEE Xplore. Restrictions apply.

to wait for the VM to be downloaded again. This time may change since it depends on the Internet speed and the size of the image being downloaded.

- Static resources on fogs can become overloaded, therefore any request of a new service at this stage will be served by the cloud. As shown in Fig. 3, the overloaded static fog and the cloud behave the same way and take 220.65s to serve the 1000 requests. Here comes the advantage of using volunteering fogs to push services using containers next to users. In our case, 97% time saving is achieved compared to the overloaded fogs and cloud use.

B. Experiment 2 - Service Deployment Including Fog Initialization

In this experiment, we study the possible scenarios that might occur in case of lack of resources to host the fog services. The possible scenarios are:

1) The orchestrator is ready, whether on a nearby or on the cloud, and no volunteering nodes are available to host the required services. In this case, the user has to wait for the joining time of the volunteer with the orchestrator.
2) The cloud plays the role of an orchestrator because of lack of fog resources. In this case, the user has to wait for the joining time for the volunteer with the cloud orchestrator.
3) The Kubeadm cluster needs to be built from scratch. In this case, the user has to wait for the whole orchestrator and fog initialization.
4) The worst case where resources are not available and requests can only be served by the cloud.

1) Experiment Setup: Kubeadm orchestrator was initialized on both the Amazon Web Service instance and Windows 7 volunteering orchestrator. Supposing that the Pi2 machine near the requesting user was the chosen fog by the orchestrator(cloud or nearby), the orchestrator sends a join command to the newly selected fog to become part of the Kubeadm, and wait until all the required pods are running properly on that fog to establish the connection with the master. After that, all the components represented in our architecture, which should run on the fog, are pushed to Pi2 volunteer. We started the experiment without any fog initialized in the area of a user requesting our service while sending a different number of requests simultaneously to the cloud until the fog is initialized and all requests are served by it. We also studied the time of creating a new orchestrator on a volunteer and joining a fog to it. We compared these scenarios with the response time of a regular cloud, hosting the service.

2) Experimental Results: In the results shown in Fig. 4, the four cases discussed are implemented and analyzed. At the beginning, all requests are served by the cloud until the fog gets ready in each of the four possible scenarios. The results of these test cases are analyzed as follows:

- In the case of having the cluster initialized on Windows7 orchestrator, the user waits for a total of 25s until the Pi2 is ready and serving as fog. This includes the time of joining the cluster and running the service.

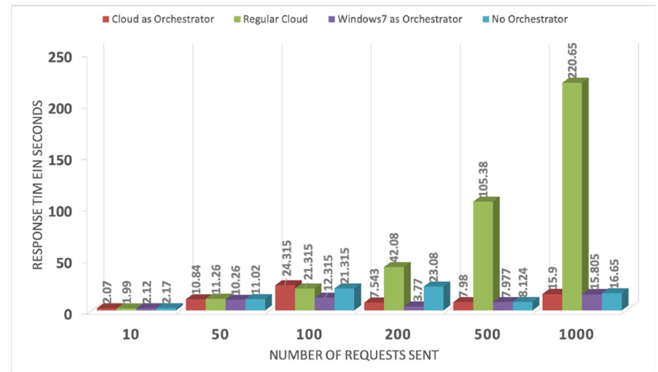

Fig. 4. Response Time of Simultaneous Requests Sent to On-Demand Fog Running On Pi2 as Fog With Cloud as Orchestrator Against Nearby Windows7 as Orchestrator, Initialized cluster from scratch on volunteers, and Regular Cloud.

TABLE II
SERVICES SELECTED FROM GOOGLE CLUSTER 2011 DATASET

Services	CPU	MEM	DISK	Priority
S1	0.12	0.2	0.2	0
S2	0.24	0.23	0.1	1
S3	0.18	0.20	0.32	0
S4	0.25	0.42	0.2	1
S5	0.31	0.15	0.24	1
S6	0.375	0.175	0.08	0

- The absence of an initialized orchestrator and lack of resources to build one, results in using the cloud as orchestrator. The user then waits for around 38s to get the Pi2 ready and running. Therefore, it is less time consuming to use a nearby volunteering orchestrator (like in the previous scenario).
- Another possibility is the presence of available resources allowing our framework to create the cluster from scratch including the orchestartor initialization, fog joining, and service preparation times. The results show that 64s are needed to prepare the cluster.

As discussed in our proposed architecture, the cluster should be created beforehand assuming that enough resources are available. To achieve such assumption, a pricing model can be proposed to motivate volunteers in joining a Kubeadm cluster. Such model is out of the scope of this paper and introduced as a future work.

C. Experiment 3.1 - Evaluation of the Container Placement Model

As previously mentioned, best case scenarios were taken in the last experiments, where we assumed that the volunteer's selection and the service placement on devices are optimal, which is not the case in real life environment. As a solution, we proposed the multi-objective container placement model. This section shows the advantage of integrating our MA in the proposed framework. Containernet [31] simulator is used, and promising results are shown.

1) Experiment Setup: In our conducted experiments, all weights are equal to 1/5 to ensure uniform influence of each objective function on the final decision. In addition, the constant C of equation 10 is equal to 10 and the priority value Pi of

Authorized licensed use limited to: Bibliothèque ÉTS. Downloaded on June 25,2021 at 03:51:42 UTC from IEEE Xplore. Restrictions apply.

TABLE III
HOSTS SELECTED FROM GOOGLE CLUSTER 2011 DATASET

Hosts	CPU	MEM	DISK	Time	Distance
H1	0.5	0.24	0.4	2000	500
H2	0.25	0.4	0.4	500	50
H3	0.25	0.26	0.2	300	40
H4	0.8	0.24	0.38	1000	25
H5	0.2	0.76	0.62	70	50
H6	0.6	0.6	0.5	60	1000
H7	1	1	1	65	20

service i is either zero or one (as shown in table II), indicating a high vs low priority to push a service. Therefore, C^{P_i} can either be 1 or 10. Moreover, the algorithm stops when it loops 1000 times or earlier when the solution converges. To evaluate our model, we build two different scenarios by passing different inputs of hosts and services to the MA, as shown in tables II, III, and IV. The input passed to the Memetic solution does not assume known topology or pre-selected fogs. This MA model produces container/service placement decisions. To visualize their effectiveness, we picked the first scenario in Table IV and compared its MA decision with a worst case selection and two outputs of a simple search algorithm that considers the first fit of an objective at a time (long time or short distance). A worst case scenario can happen when the selection has the longest distance from the user and the shortest time availability. In [16], the objective of maximizing time availability constraint of the fog is not considered. Therefore, we show the importance of considering this objective function in our case by replicating their work and removing the contribution and effect of F_3 in equation 11. We refer to this solution as the first fit on a short distance.

To test the output of the MA in a real scenario, we used Containernet simulator. For our simulation, we get the output of the MA as the assignment of services on hosts, where services are images pulled by each Docker host when the simulation starts. In Docker, we can assign CPU, Memory, and Disk constraints. Based on the services needed, we chose suitable hosts. We added users as hosts and linked all of them with switches. The link delay is specified based on the distance of each service/container to the host. Links up or links down feature in containernet simulates the reachability of a fog. Accordingly, we can disconnect a user from a fog device while the simulation is running. Accessing the terminal of each host is also possible to communicate with fogs and ask for services. Whenever the link is down, the requests go to the cloud host with the highest networking delay on the link.

The aim is to keep a record of the time it takes to respond to each user's requests of service hosted on fog volunteers by changing the number of requests. Results are obtained to show the relevance of our main approach combined with the MA solution for our proposed multi-objective optimization problem formulation. First, the search algorithm finds the first host with the highest time availability and assign as many services to it as it can handle. Similarly, for the distance factor, the available volunteer closest to the user is considered first in the decision as in [16]. The computation time of the MA depends on how many generations and individuals per generation we want as possible solutions.

TABLE IV
TWO SCENARIOS COMPOSED OF HOSTS AND SERVICES

Scenarios	Available Hosts	Requested Services
Scenario 1	H1, H2, H3, H4, H5, H6, H7	S1, S2, S3
Scenario 2	H1, H2, H3, H7	S1, S2, S3, S4, S5, S6

To get a list of services and volunteers with their requirements and capacities, we referred to Google Cluster Usage Traced dataset of 2011 [32]. This data provides a list of tasks to be pushed on available hosts. The tasks are running on hosts in an isolated way using containers. The Google Cluster dataset states the actual capacity of each host, vs. the resources requirements of each task to be executed on a host. Services are tasks to be executed inside containers on a host inside a cluster which is the volunteer in our case.

To use this data, we randomly sample 10 services and 6 hosts, as shown in tables II and III. Adding random time in seconds and distance in meters to the sampled data, we get the results of table III that symbolizes the input of volunteers capacities. Similarly, the priority level is added to table II, where 0 means a low priority. The sum of CPU, Memory, or Disk requirements sums up to 1 within the same cluster.

Flask framework is used to implement the Web service mentioned previously using Python. Our implementation for the MA follows the same structure of [15]. Our code is written in C programming language.

2) Experimental Results: In each Scenario, we have a set of input S_i needed to be pushed and a list H_j of volunteers. The MA decision model produces a placement output of K_{ij}. We varied the number of services and hosts, as well as services requirements and hosts capacities, as shown in Table IV. The aim is to show how our MA placement algorithm produces efficient results compared to existing work.

The memetic output of scenario 1 is as follows. S_1 is assigned to H_3, S_2 is assigned to H_4, and S_3 is assigned to H_2. In table III, H_4 has the highest value of time availability and the lowest value of the distance from users. This interprets why H_4 got assigned the service with high priority. H_4 will exceed its allowed resources to run more containers. If we compare all other hosts, we can see that H_2 and H_3 have a better value in term of both high availability and low distance. Our objective is to maximize survivability and minimize distance. This is why H_2 and H_3 are the best choices for the remaining services that have low priority. Furthermore, all services are pushed, including S_2 with priority equal to one, and we used the minimum number of volunteers possible to host all the required services. Moreover, all the constraints of our multi-objective problem are respected. The memetic output of scenario 2 is as follows. S_1, S_3, and S_6 with low priority are pushed to H_7. QoS is preserved, where all services with priority one are pushed first to hosts having longer time serving and a short distance from the user. S_2 is pushed to H_2, S_4 is pushed to H_3, and S_5 is pushed to H_1. All services are pushed, and all constraints are met.

Performance Analysis: To emphasize on the effectiveness of our decision model, we decided to verify the results of scenario one compared to the worst case scenario and two

Authorized licensed use limited to: Bibliothèque ÉTS. Downloaded on June 25,2021 at 03:51:42 UTC from IEEE Xplore. Restrictions apply.

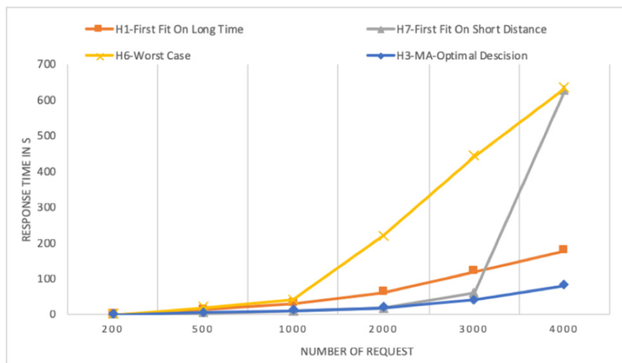

Fig. 5. Response time of the different number of requests sent simultaneously to S1 hosted on h6, h7, h1, and h5 that are the output of worst-case scenario, first fit with the lowest distance, first fit with highest time availability, and MA decision.

random search outputs. The response time of S_1 hosted on the decided volunteering fogs with respect to the number of requests sent from the user are reported in Fig. 5.

Results show four different user experiences based on the decision of service S_1 placement on the available seven hosts. The worst case scenario assigns S_1 to H_6, which has a low time availability and is 1 km away from the user. It starts with a response time of 0.123 for one request, which shows the networking delay caused by that distance. H_6 serving time is 60 seconds. After the 60s, we dropped the link between the user and H_6 in containernet to simulate its non-availability. As a consequence, we can see that the packets are received with a higher delay, which means the user is served by the cloud. H_7 as the selecting host is an output of a first fit based on shortest distance with low time availability which corresponds to the genetic solution proposed in [16]. Similarly, H_7 serves its 65s, and the requests continue their flow to the cloud after that. For H_1, it is a case where the first fit is on the host having the highest time serving. However, the distance is high, and the delay affects the performance as shown in the above image.

A major gain is achieved if we compare the results of the mentioned solutions to the MA decision, which select H_3 as the best volunteer to host S_1. Moreover, to show the advantage of our MA on the user experience, scenario one is implemented using containernet and compared to a similar work in [16]. Fig. 6 shows the response time of each service after changing the number of requests sent simultaneously. The results illustrate the advantage of using our MA solution by drastically decreasing the response time compared to results of [16] where survivability factor is not considered, and all services are pushed to H_7 which leaves the cluster after 65s. The slight differences in response time between the three services is because of the distance difference to the user. All services continue to be served by the fogs because the survivability factor is large enough. After 300s H_3 stops serving, and the requests to S_1 will either go to the cloud, or the orchestrator will prepare another fog based on the available ones decided by the MA.

D. Experiment 3.2 - Proof of Scalability and Near Optimality

This experiment shows the abilty of our Memetic solution to scale and converge to near optimal solutions given a large

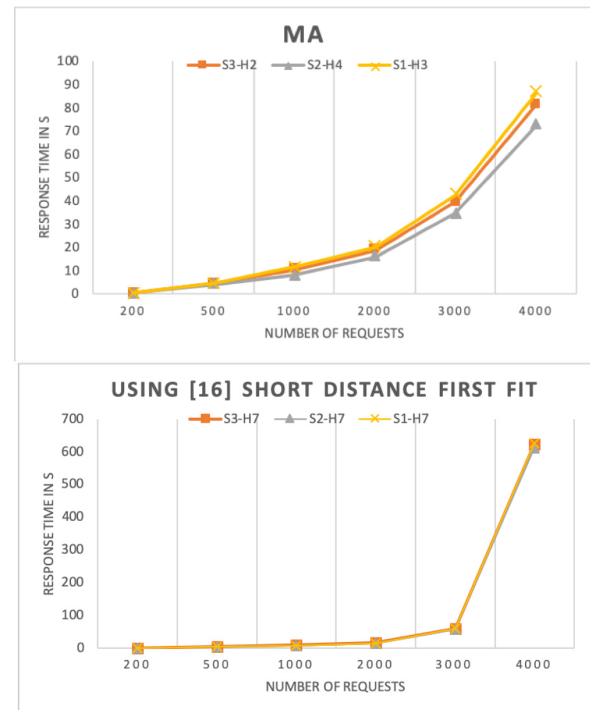

Fig. 6. Response time of different number of requests sent simultaneously to services S_1, S_2, S_3 of Scenario 1 that are hosted on H_3, H_4, and H_2 respectively based on the MA decision and all on H_7 in case of [16].

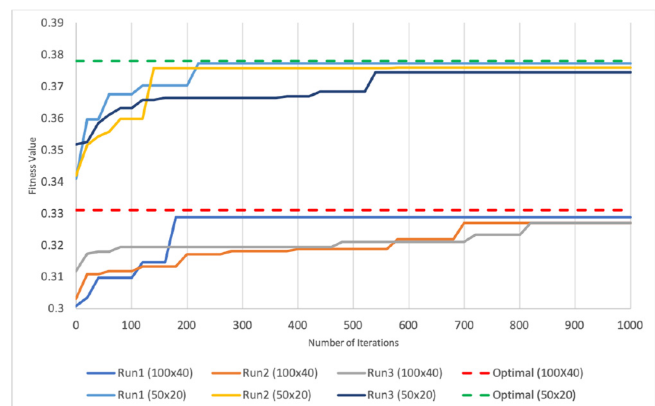

Fig. 7. Near Optimal Solutions for 100x40 and 50x20 Input Sizes.

input size, which is the case in real life scenarios. 100x40 and 50x20 input sizes are considered, where AxB means A containers to B hosts. The dataset is selected randomly from the Google Trace dataset. In our simulation, individuals' size is 100 and the number of generations (iterations) is 1000. For each input size, we ran the algorithm 10 times and plotted the best 3 solutions for each input (Fig. 7). The output is a normalized combination of all objective functions with equal weights of 1/5. The optimal solution is retrieved using an exhaustive search.

The 100x40 and 50x20 inputs are considered relatively large compared to the two scenarios of table IV. For such inputs and as shown in Fig. 7, our algorithm shows the ability to scale and converge while approaching the optimal solution.

Authorized licensed use limited to: Bibliothèque ÉTS. Downloaded on June 25,2021 at 03:51:42 UTC from IEEE Xplore. Restrictions apply.

In another experiment where small input is considered, the algorithm proves to reach the exact optimal solution.

VIII. Conclusion

The unavailability of fog devices to serve IoT users and the ability to deploy, update, and remove the services running on fogs are major issues and challenges affecting the benefits of fog architecture. In this paper, we addressed the aforementioned limitations by proposing a novel approach allowing dynamic on-demand creation of fogs on any volunteering device where micro-services are deployed on the fly. The framework includes all the modules needed for realizing the proposed scheme. The containerization technology Docker orchestrated using Kubeadm is used. Moreover, a heuristic model based on Memetic Algorithm is elaborated for solving the multi-objective fog selection and container/service placement problem. Our experimental results illustrate clearly the advantages and contributions of our solutions. The first two experiments illustrate major improvement compared to existing work while exploring the feasibility and efficiency of fog formation near the user. Moreover, the last experiment offered major support for adopting the proposed scheme in a real-life environment. As future directions, we are currently developing a security model to secure deployment of services on volunteering fogs benefiting from our architecture. We also aim to build a pricing model that motivates volunteers to join our framework. Another direction is to work on an efficient caching mechanism that can minimize the number of services deployments, as well as fog resources usage.

References

[1] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the Internet of Things," in *Proc. 1st Ed. MCC Workshop Mobile Cloud Comput.*, 2012, pp. 13–16.

[2] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Philadelphia, PA, USA, 2015, pp. 171–172.

[3] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, p. 2, 2014.

[4] D. Jaramillo, D. V. Nguyen, and R. Smart, "Leveraging microservices architecture by using docker technology," in *Proc. SoutheastCon*, Norfolk, VA, USA, 2016, pp. 1–5.

[5] G. Sayfan, *Mastering Kubernetes*. Birmingham, U.K.: Packt, 2017.

[6] P. Moscato, C. Cotta, and A. Mendes, "Memetic algorithms," in *New Optimization Techniques in Engineering*, Heidelberg, Germany: Springer, 2004, pp. 53–85.

[7] R. K. Naha *et al.*, "Fog computing: Survey of trends, architectures, requirements, and research directions," *IEEE Access*, vol. 6, pp. 47980–48009, 2018.

[8] 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, 1st Quart., 2018.

[9] M. A. Salahuddin, A. Al-Fuqaha, and M. Guizani, "Software-defined networking for RSU clouds in support of the Internet of vehicles," *IEEE Internet Things J.*, vol. 2, no. 2, pp. 133–144, Apr. 2015.

[10] P. Bellavista and A. Zanni, "Feasibility of fog computing deployment based on docker containerization over RaspberryPi," in *Proc. 18th Int. Conf. Distrib. Comput. Netw.*, 2017, p. 16.

[11] H.-J. Hong, P.-H. Tsai, and C.-H. Hsu, "Dynamic module deployment in a fog computing platform," in *Proc. 18th Asia Pac. Netw. Oper. Manag. Symp. (APNOMS)*, Kanazawa, Japan, 2016, pp. 1–6.

[12] R. Morabito, I. Farris, A. Iera, and T. Taleb, "Evaluating performance of containerized IoT services for clustered devices at the network edge," *IEEE Internet Things J.*, vol. 4, no. 4, pp. 1019–1030, Aug. 2017.

[13] C. Pahl, S. Helmer, L. Miori, J. Sanin, and B. Lee, "A container-based edge cloud PaaS architecture based on Raspberry Pi clusters," in *Proc. IEEE Int. Conf. Future Internet Things Cloud Workshops (FiCloudW)*, Vienna, Austria, 2016, pp. 117–124.

[14] C. Wöbker, A. Seitz, H. Mueller, and B. Bruegge, "Fogernetes: Deployment and management of fog computing applications," in *Proc. IEEE/IFIP Netw. Ope. Manag. Symp. (NOMS)*, 2018, pp. 1–7.

[15] F. López-Pires and B. Barán, "Many-objective virtual machine placement," *J. Grid Comput.*, vol. 15, no. 2, pp. 161–176, 2017.

[16] O. Skarlat, M. Nardelli, S. Schulte, M. Borkowski, and P. Leitner, "Optimized IoT service placement in the fog," *Service Oriented Comput. Appl.*, vol. 11, no. 4, pp. 427–443, 2017.

[17] H. Tout, C. Talhi, N. Kara, and A. Mourad, "Selective mobile cloud offloading to augment multi-persona performance and viability," *IEEE Trans. Cloud Comput.*, vol. 7, no. 2, pp. 314–328, Apr.–Jun. 2016.

[18] H. Tout, C. Talhi, N. Kara, and A. Mourad, "Smart mobile computation offloading: Centralized selective and multi-objective approach," *Expert Syst. Appl.*, vol. 80, pp. 1–13, Sep. 2017.

[19] T. Dbouk, A. Mourad, H. Otrok, H. Tout, and C. Talhi, "A novel ad-hoc mobile edge cloud offering security services through intelligent resource-aware offloading," *IEEE Trans. Netw. Service Manag.*, vol. 16, no. 4, pp. 1665–1680, Dec. 2019.

[20] M. Sookhak *et al.*, "Fog vehicular computing: Augmentation of fog computing using vehicular cloud computing," *IEEE Veh. Technol. Mag.*, vol. 12, no. 3, pp. 55–64, Sep. 2017.

[21] V. Kochar and A. Sarkar, "Real time resource allocation on a dynamic two level symbiotic fog architecture," in *Proc. 6th Int. Symp. Embedded Comput. Syst. Design (ISED)*, 2016, pp. 49–55.

[22] X. Hou, Y. Li, M. Chen, D. Wu, D. Jin, and S. Chen, "Vehicular fog computing: A viewpoint of vehicles as the infrastructures," *IEEE Trans. Veh. Technol.*, vol. 65, no. 6, pp. 3860–3873, Jun. 2016.

[23] M. Grinberg, *Flask Web Development: Developing Web Applications With Python*. Sebastopol, CA, USA: O'Reilly, 2018.

[24] H. Sami and A. Mourad, "Towards dynamic on-demand fog computing formation based on containerization technology," in *Proc. Int. Conf. Comput. Sci. Comput. Intell. (CSCI)*, 2019, pp. 960–965.

[25] H. Zeng, B. Wang, W. Deng, and W. Zhang, "Measurement and evaluation for docker container networking," in *Proc. Int. Conf. Cyber Enabled Distrib. Comput. Knowl. Disc. (CyberC)*, Nanjing, China, 2017, pp. 105–108.

[26] E. Falkenauer, "A hybrid grouping genetic algorithm for bin packing," *J. Heuristics*, vol. 2, no. 1, pp. 5–30, 1996.

[27] P. M. Pardalos, I. Steponavičė, and A. Žilinskas, "Pareto set approximation by the method of adjustable weights and successive lexicographic goal programming," *Optim. Lett.*, vol. 6, no. 4, pp. 665–678, 2012.

[28] F. López-Pires, B. Barán, A. Amarilla, L. Benítez, R. Ferreira, and S. Zalimben, "An experimental comparison of algorithms for virtual machine placement considering many objectives," in *Proc. 9th Latin America Netw. Conf.*, 2016, pp. 1–8.

[29] C. A. Coello Coello, G. B. Lamont, and D. A. Van Veldhuizen, *Evolutionary Algorithms for Solving Multi-Objective Problems*, vol. 5. New York, NY, USA: Springer, 2007,

[30] R. M. Ramadan, S. M. Gasser, M. S. El-Mahallawy, K. Hammad, and A. M. El Bakly, "A memetic optimization algorithm for multi-constrained multicast routing in ad hoc networks," *PLoS ONE*, vol. 13, no. 3, 2018, Art. no. e0193142.

[31] M. Peuster, H. Karl, and S. Van Rossem, "MeDICINE: Rapid prototyping of production-ready network services in multi-PoP environments," in *Proc. 2016 IEEE Conf. Netw. Function Virtualization Softw. Defined Netw. (NFV-SDN)*, Nov. 2016, pp. 148–153, doi: 10.1109/NFV-SDN.2016.7919490.

[32] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: Format+ schema," Mountain View, CA, USA, Google Inc., White Paper, pp. 1–14, 2011.

Hani Sami is a Research Assistant with Lebanese American University. The topics of his research include fog computing, vehicular fog computing, smart vehicles, and reinforcement learning.

Azzam Mourad is currently an Associate Professor of computer science with Lebanese American University and also an Affiliate Associate Professor with the Software Engineering and IT Department, École de Technologie Supérieure, Montreal. He has served/serves as an Associate Editor for the IEEE Open Journal of the Communications Society, *IET Quantum Communication*, and the IEEE Communications Letter, the General Chair of IWCMC2020, the General Co-Chair of WiMob2016, and the track chair, a TPC member, and a reviewer of several prestigious journals and conferences.

Authorized licensed use limited to: Bibliothèque ÉTS. Downloaded on June 25,2021 at 03:51:42 UTC from IEEE Xplore. Restrictions apply.