A Framework for Optimization, Service Placement, and Runtime Operation in the Fog

Olena Skarlat, Vasileios Karagiannis, Thomas Rausch, Kevin Bachmann, Stefan Schulte
Distributed Systems Group, TU Wien
Email: {o.skarlat, v.karagiannis, t.rausch, s.schulte}@infosys.tuwien.ac.at, kevin.bachmann@gmx.at

***Abstract*—Fog computing provides a paradigm for executing Internet of Things services. Enabling the coordinated cooperation among computational, storage, and networking resources in the fog can be challenging due to the volatility of resources. For this reason, we design an architecture and implement a representative framework called *FogFrame* that defines the necessary communication mechanisms for instantiating and maintaining service execution in the fog. To evaluate our approach, we conduct a series of experiments that show how service placement, deployment, and execution is performed by the framework, and how the framework operates at runtime, i.e., adapts to changes in the available resources, balances the workload and recovers from resource failures and overloads.**

***Keywords*-fog computing; fog computing framework, internet of things, service placement, service deployment**

I. Introduction

The advent of the Internet of Things (IoT) together with cloud technologies enables small- and large-scale applications for various domains, e.g., smart cities, smart grids, smart healthcare [1]–[3]. However, the decentralized nature of the IoT does not match the centralized structure of the cloud which leads to high link delays and low data transfer rates [4]. Furthermore, the computational resources offered by IoT devices at the edge of the network are often neglected, although these resources could be used for data processing [5]. Therefore, decentralized processing of data by devices at the edge of the network in combination with the advantages of cloud and virtualization technologies has been named as a promising approach to account for delay-sensitive, bandwidth-efficient, and resilient services in the IoT [6]. This distributed computing paradigm is known as *fog computing* [7].

To realize fog computing, it is necessary to introduce an ecosystem which involves the different computational resources offered in the cloud and at the edge of the network [8], in order to execute distributed IoT applications efficiently. This ecosystem has to provide the means for exploiting such a heterogeneous resource pool (also referred to as a *fog landscape*). To this end, we design and implement the fog computing framework *FogFrame*. FogFrame provides the means to create a real-world fog testbed based on single-board computers, i.e., Raspberry Pis, providing functionalities to establish and manage a fog landscape and to execute IoT applications. In this paper, we explicitly show how a FogFrame-based fog landscape is formed and maintained, how it evolves at runtime, and how IoT applications are submitted, processed, and executed in FogFrame.

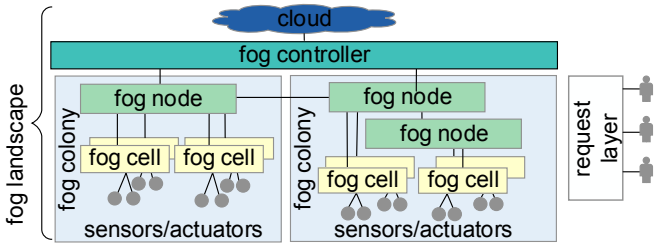

Figure 1. High-level architecture of a fog landscape

The main contribution of this paper is the FogFrame framework, which is novel in that: (i) it defines and implements functionalities that have been previously introduced in a conceptual architecture of a fog computing framework, i.e., establishment and maintenance of a fog landscape, and application management [9], [10], (ii) it implements two heuristic algorithms for service placement in the fog, namely a first-fit algorithm and a genetic algorithm, and (iii) it introduces mechanisms for adapting to dynamic changes in the fog landscape and for recovering from overloads and failures.

The remainder of this paper is organized as follows: First, we present background information on fog landscapes (Sect. II). Next, we describe how to configure a fog landscape in FogFrame (Sect. III), how to perform application management (Sect. IV), and how the fog landscape evolves at runtime (Sect. V). Afterwards, we evaluate the performance of FogFrame (Sect. VI). Finally, we overview the state-of-the-art works (Sect. VII) and conclude the paper giving insights into future work (Sect. VIII).

II. Background

Since fog computing is a recent research topic, there are different definitions of 'fog' and 'fog computing'. For the purposes of our work, we use the notion of fog landscapes which follows the OpenFog consortium [8]. The high-level architecture of a fog landscape consists of the following components [10] (Fig. 1):

Fog cells represent virtual resources in IoT devices connected to sensors and actuators. These virtual resources are used for deploying and executing arbitrary IoT services.

Fog nodes are fog cells that implement additional functionalities and act as access points for other fog cells. They perform resource provisioning and deployment of services. Their own resources are also considered in the resource pool for service placement. Fog nodes can also connect to other fog nodes in a hierarchical manner (see Fig. 1) resulting in larger resource availability.

Fog colonies consist of: (i) sensors and actuators connected to fog cells and (ii) fog cells connected to a fog node. Each fog colony has one *head fog node*, i.e., the fog node highest in the hierarchy inside the colony. Fog colonies act as dynamic micro data centers composed of distributed resources used for deploying services.

The *fog controller* is a central component used for the discovery of new resources, i.e., it receives join-requests by new fog cells and nodes, and responds with the required information to join the fog landscape. The fog controller also provisions virtual resources in the cloud, i.e., virtual machines (VMs) used as additional computational power to execute services. There is only one fog controller in each fog landscape which is placed in the cloud if the fog landscape intends to host large-scale applications or at the edge for localized IoT scenarios.

The *cloud* represents virtual resources in data centers.

Fog landscapes define the architecture for deploying and executing IoT applications. An *application* starts with input from sensors or other data sources and includes all the processing required until output commands are executed by the corresponding actuators or other data sinks. Thus, an application is a set of services needed for achieving a certain goal. Each application is submitted for execution through an *application request* initiated by users or external applications (denoted as *request layer* in Fig. 1). An application request containing information about the involved services is submitted to an arbitrary fog node, which distributes the services in the colony and the cloud. A *service request* is the request for deployment of a certain executable of a *service image* which implements the corresponding service. If this executable is deployed and running, it is called a *service*. If all services are deployed, the application can be executed. This application model is discussed thoroughly in [11]. Representative examples of this model are, e.g., stream processing applications [7], [12], [13].

III. Fog Landscape Configuration

In the following, we provide an answer to an important question of how to create a fog landscape, establish and maintain communication within it. We begin with technical assumptions to clearly delineate our work, afterwards, we provide detailed workflows.

We assume that all computational resources in the fog are virtualized [1]. In the cloud, services are placed in VMs, e.g., by the means of Docker containers. At the edge, using VMs to provision resources is inefficient due to the high start-up times and significant resource consumption which are unfit for resource- and delay-sensitive IoT applications [7]. Thus, IoT devices are assumed to be using a container-compatible operating system that can be used for the execution of containerized services [5]. Another assumption is that all fog nodes and cells are able to provide location coordinates. In addition, fog nodes integrate a proximity range used for accepting fog cells, i.e., if the cell's coordinates are within this range, the cell joins the corresponding fog node. Finally, all devices are assumed to be pre-configured with the address of the fog controller which is used for joining the fog landscape. Below, we describe how fog nodes and cells join a fog landscape, i.e., how they form colonies, and how colonies establish communication with each other for delegating application execution.

A. Creating a Fog Colony

A colony is established if a new fog node joins the fog landscape. A join-request including the fog node's location coordinates and range is sent from the new fog node to the fog controller. The controller maintains global knowledge of addresses as well as corresponding coordinates and ranges of all the fog nodes. If the new fog node is within the range of a previously joined fog node, the new fog node joins the landscape as a fog node at the lower tier in the hierarchy as depicted in Fig. 1. Otherwise, the joining fog node becomes the head of a new colony. When a fog cell requests to join, the controller responds with the address of the fog node which has a range that includes the coordinates of this cell. If the cell's coordinates are within the range of multiple fog nodes, the controller selects the one closest to the cell based on Euclidean distance. Cells are only able to join a landscape if their coordinates are within the range of a fog node.

B. Communication Between Fog Colonies

The head fog node of each colony also maintains the address of the head fog node of one neighbor colony. This address is sent to each head fog node upon joining by the controller. The controller selects this address based on proximity, i.e., minimum Euclidean distance calculated using the coordinates of the head fog nodes. The neighbor fog colony could also be selected based on different criteria, e.g., Quality of Service (QoS) statistics implemented in each fog colony and stored in the fog controller. Such statistics may include deployment delays, service execution times, resource capacities etc. In FogFrame, we choose the proximity criterion because link delays between devices may affect the deployment times of services [9].

IV. Application Management

This section presents the mechanisms implemented by FogFrame to share service images and to place and deploy services.

A. Sharing Service Images

A fog landscape consists of both cloud and edge resources which operate on different computing architectures [5]. Therefore, the service images of each application have to be compiled for the processor architecture they are intended to be executed on. To manage this heterogeneity, FogFrame implements two different solutions for sharing service images. Service images that are intended to be executed in the cloud are pushed to an online repository. In this case, service images are downloaded via a link address included in the service request. Service images intended to be executed at the edge are pushed to a fog node along with the initial application request. Since applications are executed in a distributed manner inside fog colonies, every device in a colony requires access to the service images. For this reason, the devices of each fog colony share the service images using a *shared service registry*. FogFrame uses a distributed key-value store as underlying data management system within each colony to enable a flexible schema for sharing the service image data among fog nodes and cells.

B. Service Placement

Application requests can be initially submitted to any fog node or cell, yet they are always forwarded to the head fog node for service placement. The head fog node resides at the top of the hierarchy in the fog colony and is aware of all the resources of the colony. After receiving an application request, the head fog node processes it to generate a set of service requests. Each service request includes a flag value indicating whether the corresponding service must be executed in the cloud or at the edge. Service requests intended to be executed in the cloud are separated and kept in a queue for being sent to the cloud. All other service requests are used as input to the placement algorithm. Since the placement algorithm needs to be aware of resource availability, the utilization of resources at the edge is monitored using a light-weight service called *host monitor*, which is automatically initiated in all fog nodes and cells. It periodically probes system resource and utilization data which are stored and used as input for the service placement and resource provisioning mechanisms.

The placement algorithm generates a *service placement plan* that consists of the following subsets of decisions: (i) which services have to be placed on fog nodes and cells of the colony, (ii) which services have to be placed on the resources of the head fog node, (iii) which services have to be deployed in the cloud, and (iv) whether the entire application has to be delegated to the neighbor fog colony.

First-fit Algorithm. The first-fit algorithm finds first most appropriate resource to deploy a service according to available resource capacities. In this algorithm [10], the resource pool of the fog colony is sorted by available resource capacities and by the types of services that can be hosted there, e.g., resources with sensor equipment can host sensing service. The algorithm iterates over each service request and the resource pool of the fog colony, and checks if the current resource satisfies the service according to its service type and utilization. If these constraints are met by a fog cell, then it is able to host a service, and the head fog node sends a deployment request to the fog cell. Upon successful deployment, the details about the deployed container are saved in the head fog node to keep track of all the deployed containers in the fog colony. The placement of services at the edge is prioritized over placement in the cloud. If the fog colony cannot host an application, the application request is sent to the neighbor colony.

Genetic Algorithm. Genetic algorithms have been shown to provide good results for service composition problems [14]. A genetic algorithm iteratively browses a large search space and finds near-optimal solutions in polynomial time [15]. The genetic algorithm is light-weight and therefore is able to run on resource-constrained devices at the edge of the network.

In the following, we briefly discuss common principles of genetic algorithms. Each iteration is a process of applying the genetic operators of selection, crossover, and mutation on a generation of possible solutions of a problem [16]. A generation consists of individuals represented by their chromosomes, which are characterized by fitness function values. The algorithm iterates until a certain stopping condition is activated. The challenge in the implementation of a genetic algorithm is (i) to create a chromosome representation corresponding to the desirable solution of the problem, (ii) to apply necessary parameters of genetic operators, (iii) to design a fitness function that would reflect the desirable optimization goal, and (iv) to construct an efficient stopping condition. The genetic algorithm applied in FogFrame is implemented as follows:

The chromosome is a vector corresponding to a service placement plan of the length of the total number of services in the requested application. Each value in this vector is an integer identifier of a resource in the fog colony, or the cloud. If a service cannot be placed to any of the devices or in the cloud, the value in the vector is 0. In this case, the application is supposed to be sent to the neighbor colony. Such vector ensures the placement of all services, i.e., all chromosomes are valid. The chromosome stores necessary data to estimate utilization of devices and response time of the application.

In this work, we use the following parameters of the operators based on pre-experiments [9]: A 80%-uniform crossover because the genes are integer values, crossover

mixing ratio of 0.5, tournament selection with the arity 2, random gene mutation with 2% mutation rate, 20% elitism rate, and a population size of 1000 individuals.

The fitness function is calculated based on the constraints of resources and QoS. We encourage a chromosome if it fulfills certain constraints, and apply penalties, if the constraints are violated [17]. We have defined three possible sets of constraints which influence the fitness function in different ways, i.e., (i) a set Ψ of constraints indicating if resource constraints of CPU, RAM and storage resources are met, (ii) a set Γ of implicit binary constraints derived from the goal function, i.e., conformance to service types, indications whether cloud or fog colony resources have to be used, and prioritization of local fog colony resources compared to the cloud and resources in the neighbor fog colony, and (iii) a set Υ constraints causing the 'death' of the chromosome, i.e., when the service types, number of containers in devices, and QoS constraints are violated.

Let c denote a chromosome. As it can be seen in (1), for constraints $\forall\beta_p \in \Psi$, if $\beta_p(c) \leq 0$, the constraints are satisfied and $\delta_{\beta_p(c)} = 0$. If $\beta_p(c) > 0$, then the constraints are not satisfied and $\delta_{\beta_p(c)} = 1$ as in (1). Similarly, for $\forall\beta_\gamma(c) \in \Gamma$, $\forall\beta_\upsilon(c) \in \Upsilon$. The sum of violated constrains in Υ is $\sum_{\beta_\upsilon \in \Upsilon} \delta_{\beta_\upsilon(c)}$.

The fitness function is calculated according to (2), where $\omega_{\beta_p(c)}$ is a weight factor of $\beta_p \in \Psi$, $\omega_{\beta_\gamma(c)}$ is a weight factor of $\beta_\gamma \in \Gamma$, and ω_p is the penalty weight factor for constraints in Υ. If constraints β_p or β_γ are satisfied in c, then $\delta_{\beta_p(c)}$ and $\delta_{\beta_\gamma(c)}$ become 0, and the according values within the first and the second terms are added to the fitness function. When the constraints are not satisfied, $\delta_{\beta_p(c)}$ and $\delta_{\beta_\gamma(c)}$ become 1, and the fitness function is decreased by those values. The third term provides the death penalty $\omega_p \sum_{\beta_\upsilon \in \Upsilon} \delta_{\beta_\upsilon(c)}$ for having the sum of violated constraints other than 0, where the penalty factor ω_p has to be big enough to forbid the worst chromosomes from being selected for crossover. During the runtime of the algorithm, the fitness function increases as less penalties are applied to the chromosomes.

$$\delta_{\beta_p(c)} = \begin{cases} 0, & \text{if } \beta_p(c) \leq 0 \\ 1, & \text{if } \beta_p(c) > 0 \end{cases} \tag{1}$$

$$F(c) = \sum_{\beta_p \in \Psi} \omega_{\beta_p}(1 - 2\delta_{\beta_p(c)}) + \sum_{\beta_\gamma \in \Gamma} \omega_{\beta_\gamma}(1 - 2\delta_{\beta_\gamma(c)}) - \omega_p \sum_{\beta_\upsilon \in \Upsilon} \delta_{\beta_\upsilon(c)} \tag{2}$$

The algorithm stops based on several conditions. First, the fitness value of the fittest individual in the generation has to be a positive number, which means that no death penalties have been applied to the chromosome. Then, if this condition is fulfilled, specific stopping conditions are checked. The first condition is variance-based, i.e., the tolerance of the fitness function is calculated by dividing the incremental variance of the fitness values by the maximum fitness value over generations [18]. We set the tolerance value of the fitness function to be less than $\epsilon = 0.01$, which is enough to obtain the solution and not to converge in local maxima. To limit unproductive search, we add an auxiliary time-based stopping condition, i.e., a limit on the number of iterations.

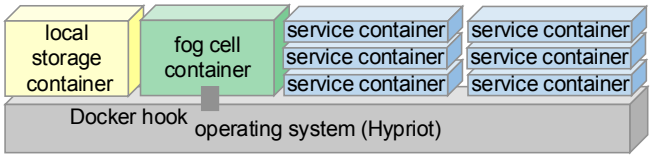

Figure 2. Deployment on a fog cell

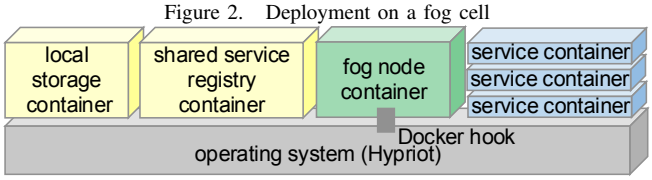

Figure 3. Deployment on a fog node

C. Service Deployment and Execution

Since the two computational environments in fog landscapes (i.e., cloud and edge) are different, deployment mechanisms for services also need to differ. In the cloud, services are deployed in Docker containers on VMs, which need to be deployed and managed. When a service request is sent to the fog controller from a fog colony and there is no deployed VM in the cloud, the fog controller leases and starts a new VM. If a VM is already running, the corresponding Docker container of the necessary service image is deployed on that VM. The containers are deployed on a cloud VM until a certain limit of containers is reached to ensure the stability of the computational environment. If there are no free resources for another container to be deployed on a VM, a new VM is leased and a container is deployed there. When the execution is finished, containers are stopped. If the VM is running with no load, the VM is stopped and the cloud resources are released again.

For fog colonies, the deployment mechanisms differ. To make it possible for fog cells and fog nodes to deploy, start, and stop further Docker containers on the host device, we make use of a *Docker hook* (see Fig. 2 and Fig. 3) [19]. This Docker hook resolves the problem of instantiating other Docker containers on the Docker runtime of the host device from inside the Docker containers of the running fog cells and fog nodes.

V. Fog Landscape Runtime

A fog landscape is dynamic in its nature, as devices may appear and disappear arbitrarily. Since failures and overloads are inevitable in such systems, the fog landscape refines the network based on periodic and event-based mechanisms at

runtime by redeploying running applications. In the following, we describe redeployment and placement replanning mechanisms for fog colonies as implemented in FogFrame.

A. Overloaded Devices

Resource utilization in all devices in a colony is polled periodically by the head fog node. If a device is identified as overloaded, i.e., it surpasses preconfigured utilization thresholds (related to CPU, RAM and storage capacities), the head fog node redeploys one random service from the overloaded device to another one inside the colony using the placement procedure. This process continues until resource utilization drops below the threshold.

B. Disconnected Devices

All devices in a colony are polled periodically by the head fog node. Devices that do not respond are considered disconnected. When a head fog node discovers a disconnected device, it redeploys the services that were running on the disconnected device (based on the service placement plan) to other devices in the fog colony using the placement procedure. If the head fog node is disconnected, a fog node lower in the hierarchy becomes the new head fog node triggered by the absence of the previous head fog node's polling. If there is no other fog node in the fog colony, the colony fails. If a neighbor fog colony is unresponsive, the head fog node requests the address of a different neighbor fog colony from the fog controller. If the fog controller is unresponsive, fog nodes are also able to accept join-requests (as long as the new device knows the fog node's address) which leads to placing the new device in the lower tier of the hierarchical structure of the colony.

C. New Devices

Each time a new fog node or cell joins a colony, it triggers the replanning mechanism executed by the head fog node. More specifically, the head fog node of the colony examines the available resources to identify resource utilization and the number of deployed services in all the devices of the colony. Then, the head fog node opportunistically redeploys services that could be executed in the colony, but are currently running in the cloud, to the new fog cell. Afterwards, services that run on devices operating at maximum capacity are scheduled for redeployment so that some of them will be deployed on the new device. Therefore, through the replanning mechanism, fog colonies react to the increase in the available resources and re-balance the workload to all the devices.

In order to briefly summarize the presented functionalities of fog cells and fog nodes, we discuss their high-level architecture (see Fig. 4). A fog cell consists of a database service which operates a local database and stores connection data, identification data, and application execution data as mentioned at the beginning of Sect. III. The communication

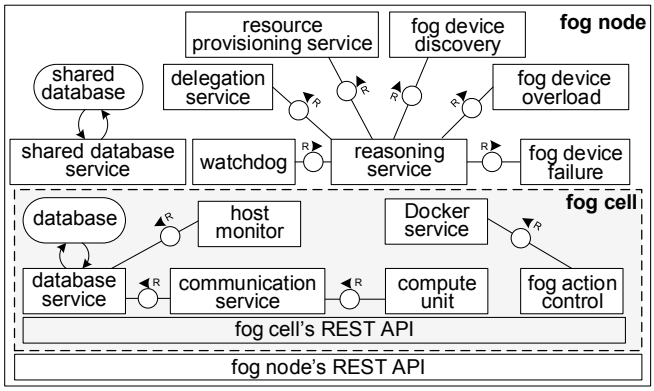

Figure 4. High-level architecture of fog cells and fog nodes.

service establishes and maintains the communication with the fog controller and a parent fog node. The compute unit executes services and transfers data between services. The fog action control service follows the orders from the head fog node of the fog colony, and deploys necessary services by the means of the Docker service. There is a dedicated host monitor service, which monitors the utilization of the fog cell. These components have been described in Sect. IV.

A fog node consists of all the components of a fog cell and provides some extensions. The extensions of the fog node are the shared database service which operates the shared database with the service registry. The watchdog monitors the utilization data of all the connected fog cells, and triggers the reasoning service to react on runtime events as was presented at the beginning of Sect. V. The delegation service sends application requests to the closest neighbor fog colony and service requests to the cloud. The reasoning service is triggered when an application request is submitted for execution to the fog node. For this, it calls the resource provisioning service, which implements a certain service placement algorithm. The reasoning service also triggers device discovery mechanism to react on new devices, and collective healing mechanisms for fog colonies to react on disconnected and overloaded devices as described in Sect. V-A to V-C, i.e., new device discovery, device overload and device failure mechanisms. To summarize, the FogFrame framework provides all necessary functionalities for application management and accounts for the volatile nature of the fog landscape.

VI. Evaluation

To evaluate FogFrame, we examine how services are distributed in the fog landscape under different arrival patterns of service requests, and how FogFrame reacts to different runtime events in the fog landscape.

A. Metrics

In order to show the results of the evaluation, we measure the following metrics: (i) deployment time per service at the

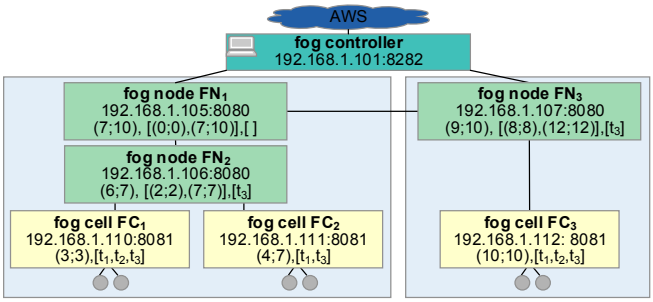

Figure 5. Evaluation setup

edge and in the cloud, (ii) time-to-recover per service due to failures, and accordingly (iii) number of redeployed services due to failures, (iv) percentages of successful recovery, and (v) time-to-redeploy a service due to overloads. In order to examine the behavior of FogFrame, we also record the workload of each device in terms of number of deployed services over time. We examine this load in a stacked form, i.e., the load in each device and the total load.

B. Experimental Setup

FogFrame is implemented using Java 8 and the Spring application framework. Services in FogFrame interact by means of REST APIs and JSON messages. The database technology used both for local databases and the shared database is Redis. The fog controller is executed on a Ubuntu 16.04 LTS VM on a standard notebook. For cloud resources, we use Amazon AWS EC2 services, specifically t2.micro VMs with CoreOS, which has a Docker environment setup by default. Fog nodes and fog cells are deployed on Raspberry Pi 3 units with the Hypriot operating system. More details about the Raspberry Pi configuration can be found in [19]. The framework is available under an open source license at Github[1].

As can be seen in Fig. 5, fog nodes FN_1 and FN_3 are connected to the fog controller. The fog colony controlled and orchestrated by FN_1 consists of a fog node FN_2, and two fog cells FC_1 and FC_2. The fog colony controlled and orchestrated by FN_3 has one connected fog cell FC_3. As data sources, temperature and humidity sensors are installed on the Raspberry Pis by the means of GrovePi sensor boards.

In the experimental setup, each application consists of a number of service requests of certain service types. For the purposes of the evaluation, we have defined and implemented three possible service types: Services of type t_1 receive data from temperature and humidity sensors and are executable only on fog cells because services of this type need sensor equipment; services of type t_2 and t_3 simulate processor load by continuously writing into a string, and are executable either in fog colonies or in the cloud. We have also developed a dedicated *cloud service* which receives sensor readings and writes them to a cloud database.

[1]https://github.com/softls/FogFrame-2.0

C. Experiments

To evaluate the behavior of FogFrame, we apply different arrival patterns of application requests, i.e., constant, pyramid, and random walk, and observe service placement. The arrival patterns are shown along with the representative results of experiments in Fig. 6. The baseline for evaluation is the uninterrupted operation of the fog landscape observed during the first ten minutes of each run of the experiment. After 10 minutes, failures are automatically introduced. Specifically, failures are generated with the probability $P(Failure) = 0.05\%$ per second, i.e., $P(Failure) = 30\%$ in 10 minutes, and recorded to be introduced in each run. Failures happen at about the 11^{th} and 16^{th} minute of each run. The device failure is simulated by stopping the fog cell application container at the chosen device. In reality, device failures can be hardware, software, or communication problems. Whenever a fog cell has a failure, we use its Raspberry Pi to deploy a new fog cell. We observe how the system redeploys services running on the disconnected fog cell and measure time-to-recover per service. The overloads may occur in the course of execution, e.g., if the CPU load is 100%. In the case of an overload, the device is still operating, however, some services have to be redeployed to release resources, e.g., using a newly joined fog cell, already existing resources, or cloud resources. We run the experiment ten times for each arrival pattern in combination with each placement algorithm and provide statistical distributions of results.

D. Results and Discussion

The evaluation results for the six combinations of the two algorithms (first-fit, genetic algorithm) and three arrival patterns (constant, pyramid, random walk) are shown in Fig. 6. During the first ten minutes of each run, in the first-fit placement, the services are placed on the fog cells to the maximum capacity. If both fog cells FC_1 and FC_2 are loaded, and a new application request arrives with services which need sensor equipment, the deployment in the current fog colony becomes impossible, and the fog node decides to delegate the application to the closest neighbor colony. In the genetic algorithm, more time is spent for the deployment of single services compared to the first-fit placement because the algorithm requests device utilization information before starting calculations. However, more application requests are delegated to the neighbor fog colony, and the load on the resources is better balanced, which allows to avoid overloads and is crucial if new services need sensor equipment.

After ten minutes, we observe how FogFrame reacts on different runtime events. The first case to consider is when a connected and successfully paired fog cell loses its connection with the fog landscape due to a device failure.

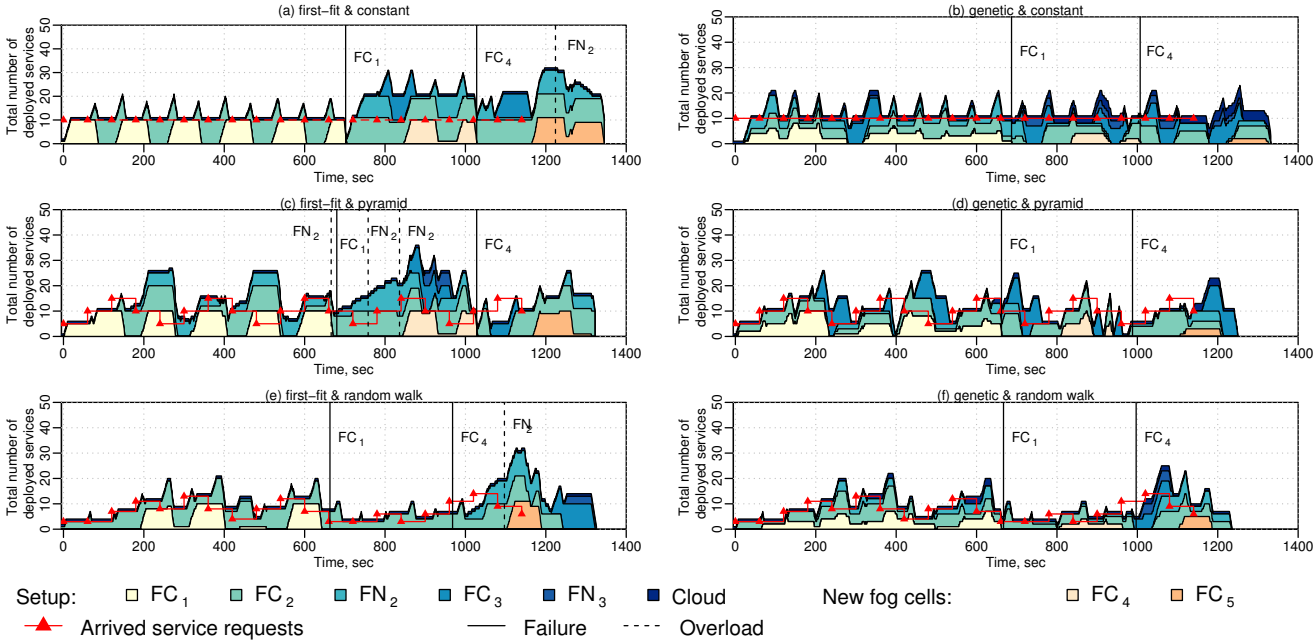

Figure 6. Results of experiments with different placement algorithms and arrival patterns

In this experiment, the device to fail is the Raspberry Pi of FC_1. In order to demonstrate how the framework reacts when a new device appears in a fog colony, its coherent mechanisms and service redeployment, a new fog cell not yet associated with the fog colony is instantiated. The Raspberry Pi of FC_1 is used to deploy new fog cells FC_4 after the first failure, and FC_5 after the second failure. In the first-fit placement, the percentage of successful recovery is lower than in the genetic algorithm placement. This happens because in the first-fit placement both fog cells are fully loaded, and some services which need sensor equipment cannot be redeployed in the fog colony. In the genetic algorithm, the fog colony's resources are not loaded to the maximum capacities, and therefore less services need to be redeployed, and consequently the fog colony has enough resources to perform such redeployment. In the course of the experiments, overloads occur mostly in FN_2 because this fog node apart from executing services also handles the communication between the connected fog cells and the cloud. In the genetic algorithm placement, the underlying fog cells and FN_2 are not fully loaded, and FN_2 has enough utilization capacities to deal with the communication.

To summarize this experiment (see Tab. I), the genetic algorithm performs better with regard to distributing service requests within the fog colony and between the fog colonies. The positive aspect in the genetic algorithm placement is that the resources in the fog landscape are not close to overload, as can be also seen in Fig. 6, giving more opportunities for newly requested services to be deployed, i.e., because of better availability of sensor equipment.

Table I
EXPERIMENT RESULTS OVERVIEW

Metrics	Algorithm	Constant	Pyramid	Random
Deployment time per service, edge (sec)	First-fit	2.41 σ =1.10	3.07 σ =0,93	3.06 σ =1.08
	Genetic	2.92 σ =1.16	3.02 σ =0.92	3.05 σ =1.17
–"– cloud (sec)	Genetic	2.78 σ =0.18	2.69 σ =0.13	3.18 σ =0.63
Number of services delegated	First-fit	26 σ =21	24 σ =21	24 σ =10
	Genetic	59 σ = 9	46 σ = 20	26 σ = 13
Number of services to recover	First-fit	21 σ =3	11 σ =6	10 σ =5
	Genetic	6 σ = 4	7 σ = 4	6 σ = 2
Time-to-recover per service (sec)	First-fit	3.88 σ =3.12	2.49 σ =1.18	1.98 σ =0.12
	Genetic	2.09 σ =0.24	2.02 σ =0.18	2.07 σ =0.10
Successful recovery (%)	First-fit	90.00 σ =12.84	89.36 σ =14.62	99.11 σ =2.36
	Genetic	100.00 σ =0.00	100.00 σ =0.00	100.00 σ =0.00
Time-to-redeploy per service (sec)	First-fit	1.80 σ =0.06	9.41 σ =2.95	3.16 σ =1.13
	Genetic	2.46 σ =0.05	2.49 σ =0.05	2.37 σ =0.17

VII. RELATED WORK

Fog computing is still an emerging computing paradigm. The conceptual and theoretical work about fog computing use cases, definitions and conceptual architectures is quite extensive, however there is a lack of concrete implemen-

tations of fog computing frameworks. In this section, we provide an overview of existing approaches and frameworks focusing on how to establish and maintain fog landscapes.

A conceptual work by Varshney and Simmhan [20] describes coordination models to be applied in fog computing. In their work, three different coordination models in fog computing are considered, i.e., (i) hierarchical, (ii) peer-to-peer, and (iii) hybrid. It is emphasized that the *hierarchical model* allows only vertical communication. The cloud is defined as the root of this hierarchy and is responsible for all the coordination, orchestration, and execution of applications. In the *peer-to-peer model*, horizontal communication is considered between different devices in the fog landscape. Such communication has to be set up by a central entity, which can be located either in the cloud or at the edge of the network, and which has a global view on the overall pool of resources. The network topology has to be maintained by the means of a distributed hash table. In the *hybrid coordination model*, the authors consider both horizontal communication between different resources as well as vertical communication to establish an ordered fog landscape infrastructure. We place our work in such a hybrid model. In FogFrame, a central entity to form and maintain a fog landscape is the fog controller. Fog colonies are interconnected and collaboratively can recover from failures and execute applications even if the fog controller is not available. With regard to the fog landscape operation, Varchney and Simmhan propose to change the coordination model depending on changes of the fog landscape at runtime.

Byers [21] provides a conceptual description of architectural imperatives for fog computing, i.e., critical requirements for fog computing architectures and techniques to resolve the mentioned problems. One of the considered imperatives is *geographic locality and control*. This imperative requires the fog landscape to control the information flow based on physical and logical boundaries, and tells that there is no valid reason to send data beyond those boundaries. Another imperative is *reliability and robustness*, which requires to maintain operation of the fog landscape. This necessitates adaptive approaches to migrate applications in response to runtime events by applying fault tolerance and redundancy scenarios at different levels of the fog landscape hierarchy. The *hierarchical organization* imperative assumes a tree-like hierarchy of the fog landscape. Another imperative is *agility* that requires fast reaction on runtime events and adapting the fog infrastructure accordingly. Last but not least, a coordination-related imperative is *scalability*, which is considered both from physical and software point of views. The fog is expected to grow continuously, and therefore the architectural design of the fog computing environment has to be adaptive to such growth. In our work, we address such imperatives, i.e., our framework is hierarchical, it allows for both horizontal and vertical communication within the fog landscape, it is adaptive and scalable, it accounts for the geographical distribution of its resource pool, and ensures collaboration between entities in the fog landscape to provide a certain level of QoS and infrastructure reliability.

Zhang et al. [22] propose a conceptual regional cooperative fog computing architecture. The authors distinguish between edge and fog layers, and a cloud center. The fog layer is coordinated by a separate *local coordination server*. The cloud center is a *super fog server*, however the coordination is an exclusive responsibility of the local coordination server. The authors focus on cooperation in the fog landscape in terms of service migration, maintaining uninterrupted service and reliability. With regard to communication, intra-fog and inter-fog communication and management is envisioned. The virtualization mechanism proposed in their work is by the means of VMs. Compared to our work, we use containers instead of VMs, since VMs are heavy-weight and therefore not suited for rather light-weight IoT devices [5]. Similarly to Zhang et al., we also do not design any coordinating fog landscape entities in the cloud. A local coordination server resembles a fog node being a head of a fog colony, however, we provide communication and collaboration between different fog colonies by the means of their corresponding fog nodes.

Tsai et al. [23] introduce an analytic fog computing platform based on TensorFlow and Kubernetes. This Raspberry Pi-based platform consists of a centralized server and fog devices. Fog devices in their work are defined as all possible resources from data centers and edge networks. The programming of the application model is performed by the means of TensorFlow. Kubernetes manages the fog landscape, identifies the available resources, and deploys containers of operators. In our work, the orchestration mechanism is not centralized as it it done with Kubernetes. Compared to Kubernetes, fog nodes are light-weight, they are deployed on the devices at the edge of the network.

In the work of He et al. [24], a multi-tier fog computing model is simulated, which consists of *dedicated fogs* and *opportunistic fogs*. Dedicated fogs are static, while opportunistic fogs are volatile and consist of various computational, storage and networking resources which may enter or leave the fog. He et al. propose the use of *fog masters* and *fog workers*. Considering the question of how to form a fog landscape, in opportunistic fogs, fog devices are joined by invitation from fog masters. Each fog has at least one fog master. An interesting aspect is that one fog can have multiple fog masters to improve reliability. However, it is not described in detail , how the coordination and management is performed by those multiple fog masters. Compared to our work, FogFrame is not a simulation framework, but a real-world fog computing framework. Fog masters resemble fog nodes in FogFrame, and fog workers resemble fog cells. We also enable multiple fog nodes in one fog colony. In order to enter the fog landscape we use self-announcement

mechanism, while He et al. propose invitations mechanism. Self-announcement is also proposed in the work of de Brito et al. [25]. They propose an IoT testbed that is built on Docker Swarm and the OpenMTC M2M Framework and consists of two main entities, a fog orchestration agent and a fog orchestrator. However, the communication between different fog orchestrators is not envisioned.

Yigitoglu et al. [26] implement a fog computing framework called Foggy. Foggy has a three-tier infrastructure, namely edge devices, network infrastructure, and cloud services. Their network infrastructure tier consists of nodes and an orchestration server. The orchestration server creates and maintains a *resource catalog* which contains data about the available resource pool, i.e., capacities, connections, and utilization. The authors also propose different strategies to promote reliability when performing deployments of services. Compared to our work, we distinguish between different types of nodes, i.e., fog cells and fog nodes. Also, our fog nodes perform orchestration in their own colonies, hence we do not have only one orchestration server, but many interconnected fog colonies.

In the work of Vögler et al. [27], a framework for dynamic generation of deployment topologies for IoT applications called DIANE is proposed. This framework monitors the deployment infrastructure, groups it according to available resources, and stores the results for later analysis. The framework dynamically deploys topologies for IoT applications by the means of a rule-base algorithm, and provides their monitoring. While in DIANE, only a rule-based algorithm is used for resource provisioning, in our work, we formulate a concrete optimization problem. We also consider in details and implement a hierarchical structure of a fog landscape focusing specifically on establishing communication between different devices in the fog landscape, and providing application management.

To summarize, the contribution of FogFrame is the coordinated control over the physical and virtual infrastructure of the resources both in the cloud and at the edge of the network. For this, FogFrame provides mechanisms to manage and support fog colonies, i.e., to establish communication within the fog landscape and handle data transfer, with the ability to optimally provision necessary computational resources and execute applications. Furthermore, it automatically discovers resources at the edge of the network and forms a fog landscape network topology and reacts on its dynamic changes during runtime. The summary of our findings from the related work is given in Tab. II.

Compared to our former work [9], [10], the work at hand provides an implemented framework. It describes specific details on runtime operation of a fog landscape. In contrast, within our former work, we were operating on a conceptual framework and primarily focused on finding solutions to the service placement problem.

Table II
RELATED WORK OVERVIEW

Work	Implement	VMs	Containers	Centralized	Distributed	Intra-fog Commun.	Inter-fog Commun.
Zhang et al. [22]		✓			✓	✓	✓
Tsai et al. [23]	✓	✓		✓		✓	
He et al. [24]					✓	✓	
de Brito et al. [25]	✓		✓		✓	✓	
Yigitoglu et al. [26]	✓		✓	✓		✓	✓
Vögler et al. [27]	✓	✓	✓	✓		✓	
FogFrame	✓	✓	✓		✓	✓	✓

VIII. CONCLUSION

In this paper, we introduced FogFrame[2], a framework for building and maintaining fog landscapes. By using this framework and Raspberry Pi computers, it is possible to build a real-world fog computing testbed for the execution of IoT applications. Apart from the necessary mechanisms for configuring a fog landscape, FogFrame provides all the application management mechanisms needed to support a full life-cycle of applications.

We evaluated FogFrame and the proposed mechanisms using the testbed. For this, we recorded the workload on different computational resources at the edge of the network and in the cloud, deployment times of services, times-to-recover and recovery rates in the case of failures and overloads. Our findings show that the framework dynamically reacts to runtime events, i.e., when new devices appear or disconnect in the fog landscape, and when devices experience failures or overloads, and performs necessary redeployments. FogFrame records information for measuring efficiency and for evaluating performance based on application execution and resource utilization monitor.

Having addressed the question of how to create a fog landscape, it is possible now to use these results as a basis for developing other communication and application management mechanisms. The framework allows easy integration of new components due to well-defined interfaces. In future work, in the infrastructure of fog landscapes, mechanisms to connect multiple different fog landscapes have to be further considered. This would allow to establish a meta-level of infrastructure for additional optimization and reconfiguration. In the application management, other service placement algorithms can be introduced to optimize service placement according to different goals, e.g., with regard to different cost models of fog resources.

ACKNOWLEDGMENT

This paper is supported by TU Wien research funds. This work is partially funded by COMET K1, FFG – Austrian Research Promotion Agency (Contract Nr. 854187), within

[2]https://github.com/softls/FogFrame-2.0

the Austrian Center for Digital Production, and the H2020 FORA project (grant No.: 764785).

REFERENCES

[1] P. Hu, S. Dhelim, H. Ning, and T. Qiu, "Survey on fog computing: architecture, key technologies, applications and open issues," *J Netw. Comput. Appl.*, vol. 98, pp. 27–42, 2017.

[2] V. Karagiannis, P. Chatzimisios, F. Vazquez-Gallego, and J. Alonso-Zarate, "A survey on application layer protocols for the Internet of Things," *Trans. on IoT and Cloud Computing*, vol. 3, no. 1, pp. 11–17, 2015.

[3] 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. Surv. Tutor.*, vol. 20, no. 1, pp. 416–464, 2018.

[4] V. Karagiannis and A. Papageorgiou, "Network-integrated edge computing orchestrator for application placement," in *13th IEEE Int. Conf. on Network and Service Management (CNSM'17)*. Tokyo, Japan: IEEE, 2017, pp. 1–5.

[5] R. Morabito, V. Cozzolino, A. Y. Ding, N. Beijar, and J. Ott, "Consolidate IoT Edge Computing with Lightweight Virtualization," *IEEE Network*, vol. 32, no. 1, pp. 102–111, 2018.

[6] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, "Fog Computing: A Platform for Internet of Things and Analytics," in *Big Data and Internet of Things: A Roadmap for Smart Environments*. Springer, 2014, vol. 546, pp. 169–186.

[7] A. V. Dastjerdi, H. Gupta, R. N. Calheiros, S. K. Ghosh, and R. Buyya, "Fog Computing: Principles, Architectures, and Applications," in *Internet of Things: Principles and Paradigms*. Morgan Kaufmann, 2016, ch. 4, pp. 1–26.

[8] "OpenFog Reference Architecture for Fog Computing," www.openfogconsortium.org/ra, 2016, online; Accessed: 14 Jun. 2018.

[9] O. Skarlat, M. Nardelli, S. Schulte, M. Borkowski, and P. Leitner, "Optimized IoT service placement in the fog," *SOCA J.*, vol. 11, no. 4, pp. 1–17, 2017.

[10] O. Skarlat, S. Schulte, M. Borkowski, and P. Leitner, "Resource Provisioning for IoT Services in the Fog," in *9th IEEE Int. Conf. on Service Oriented Computing and Applications (SOCA'16)*. Hong Kong, China: IEEE, 2016, pp. 32–39.

[11] O. Skarlat, M. Nardelli, S. Schulte, and S. Dustdar, "Towards QoS-aware Fog Service Placement," in *1st IEEE Int. Conf. on Fog and Edge Computing (ICFEC'17)*. Madrid, Spain: IEEE, 2017, pp. 89–96.

[12] V. Cardellini, V. Grassi, F. L. Presti, and M. Nardelli, "On QoS-aware scheduling of data stream applications over fog computing infrastructures," in *2015 IEEE Symposium on Computers and Communication (ISCC'15)*. Larnaca, Cyprus: IEEE, 2015, pp. 271–276.

[13] C. Hochreiner, M. VÃűgler, S. Schulte, and S. Dustdar, "Cost-efficient enactment of stream processing topologies," *PeerJ Comput. Sci.*, vol. 3, no. e141, pp. 1–36, 2017.

[14] M. Xu, W. Tian, and R. Buyya, "A survey on load balancing algorithms for virtual machines placement in cloud computing," *Concurr. Comput.*, vol. e4123, pp. 1–16, 2017.

[15] Z. Ye, X. Zhou, and A. Bouguettaya, "Genetic Algorithm Based QoS-Aware Service Compositions in Cloud Computing," in *16th Int. Conf. Database Systems for Advanced Applications (DASFAA'11)*. Hong Kong, China: Springer, 2011, pp. 321–334.

[16] D. Whitley, "A Genetic Algorithm Tutorial," *Stat. Comput.*, vol. 4, pp. 65–85, 1994.

[17] Özgür Yeniay, "Penalty Function Methods for Constrained Optimization with Genetic Algorithms," *Math. Comput. Appl.*, vol. 10, no. 1, pp. 45–56, 2005.

[18] D. Bhandari, C. A. Murthy, and S. K. Pal, "Variance As a Stopping Criterion for Genetic Algorithms with Elitist Model," *Fundam. Inf.*, vol. 120, no. 2, pp. 145–164, 2012.

[19] K. Bachmann, "Design and Implementation of a Fog Computing Framework," Master's thesis, Vienna University of Technology (TU Wien), Vienna, Austria, 2017.

[20] P. Varshney and Y. Simmhan, "Demystifying Fog Computing: Characterizing Architectures, Applications and Abstractions," in *2017 IEEE 1st Int. Conf. on Fog and Edge Computing (ICFEC)*. Madrid, Spain: IEEE, 2017, pp. 115–124.

[21] C. C. Byers, "Architectural Imperatives for Fog Computing: Use Cases, Requirements, and Architectural Techniques for Fog-Enabled IoT Networks," *IEEE Commun. Mag.*, vol. 55, no. 8, pp. 14–20, 2017.

[22] W. Zhang, Z. Zhang, and H. C. Chao, "Cooperative Fog Computing for Dealing with Big Data in the Internet of Vehicles: Architecture and Hierarchical Resource Management," *IEEE Commun. Maga.*, vol. 55, no. 12, pp. 60–67, 2017.

[23] P. H. Tsai, H. J. Hong, A. C. Cheng, and C. H. Hsu, "Distributed Analytics in Fog Computing Platforms using Tensorflow and Kubernetes," in *19th Asia-Pacific Network Operations and Management Symposium (APNOMS'17)*, Seoul, Korea, 2017, pp. 145–150.

[24] J. He, J. Wei, K. Chen, Z. Tang, Y. Zhou, and Y. Zhang, "Multi-tier Fog Computing with Large-scale IoT Data Analytics for Smart Cities," *IEEE Internet Things J.*, vol. 5, no. 2, pp. 677–686, 2018.

[25] M. de Brito, S. Hoque, T. Magedanz, R. Steinke, A. Willner, D. Nehls, O. Keilsa, and F. Schreiner, "A service orchestration architecture for fog-enabled infrastructures," in *2nd Int. Conf. on Fog and Mobile Edge Computing (FMEC'17)*, Valencia, Spain, 2017, pp. 127–132.

[26] E. Yigitoglu, M. Mohamed, L. Liu, and H. Ludwig, "Foggy: A Framework for Continuous Automated IoT Application Deployment in Fog Computing," in *IEEE Int. Conf. on AI Mobile Services (AIMS'17)*. Honolulu, Hawaii, USA: IEEE, 2017, pp. 38–45.

[27] M. Vögler, J. Schleicher, C. Inzinger, and S. Dustdar, "Optimizing Elastic IoT Application Deployments," *Trans. Serv. Comput.*, vol. PP, no. 99, pp. 1–14, 2016.