MockFog: Emulating Fog Computing Infrastructure in the Cloud

Jonathan Hasenburg, Martin Grambow, Elias Grünewald, Sascha Huk, David Bermbach
TU Berlin & Einstein Center Digital Future
Mobile Cloud Computing Research Group
{jh, mg, egr, shu, db}@mcc.tu-berlin.de

***Abstract*—Fog computing is an emerging computing paradigm that uses processing and storage capabilities located at the edge, in the cloud, and possibly in between. Testing fog applications, however, is hard since runtime infrastructures will typically be in use or may not exist, yet.**

In this paper, we propose an approach that emulates such infrastructures in the cloud. Developers can freely design emulated fog infrastructures, configure their performance characteristics, and inject failures at runtime to evaluate their application in various deployments and failure scenarios. We also present our proof-of-concept implementation MockFog and show that application performance is comparable when running on MockFog or a small fog infrastructure testbed.

***Index Terms*—Programming models, abstractions, and software engineering for fog computing; Fog computing applications and experiences**

I. Introduction

Fog computing is an emerging computing paradigm that uses processing and storage capabilities located at the edge, in the cloud, and possibly in between to deal with increasing amounts of IoT data but also to address latency and privacy requirements from use cases such autonomous driving, 5G, and eHealth [1]–[3].

However, even though fog computing has many advantages, there are currently only a few fog applications and "commercial deployments are yet to take off" [4]. Arguably, the main adoption barrier is the deployment and management of physical infrastructure, particularly at the edge, which is in stark contrast to the ease of adoption in the cloud [1].

In the lifecycle of a fog application, this is not only a problem when running and operating a production system – it is also a challenge in application testing: While basic design questions can be decided using simulation, e.g., [5]–[7], there comes a point when a new application needs to be tested in practice. The physical fog infrastructure, however, will typically be available for a very brief period of time only: in between having finished the physical deployment of devices and before going live. Before that period, the infrastructure presumably does not exist and afterwards its full capacity is used in production. Without an infrastructure to run more complex integration tests or to test fault-tolerance in wide area deployments, however, the application developer is left with guesses, (very small) local testbeds, and simulation.

In this paper, we propose to emulate fog infrastructures in the cloud: cloud machines are trivial, the challenging ones are the non-cloud machines. In an emulated fog environment, edge machines are deployed in the cloud as well which is then configured to closely mimic the real (or planned) fog infrastructure. By using basic information on network performance and failure rates, either obtained from the production environment or based on expectations and experiences with other applications, interconnections between emulated fog machines can be manipulated to show similar characteristics. Likewise, performance measurements from real fog machines can be used to determine resource limits on cloud-deployed Docker[1] containers. This way, fog applications can be dockerized and fully deployed in the cloud while experiencing comparable performance and failure characteristics as a real fog deployment[2].

For this purpose, we make the following contributions.

- We propose MockFog, a system design for the emulation of fog computing infrastructure in arbitrary cloud environments and discuss how the infrastructure setup process integrates into an application engineering process.
- We present our proof-of-concept implementation which allows developers to quickly model, configure, and deploy an emulated fog infrastructure on either Amazon EC2[3] or OpenStack[4].

While testing in an emulated fog will never be as "good" as in a real production fog environment, it is certainly better than simulation-based evaluation only. Moreover, it allows application engineers to test arbitrary failure scenarios, future runtime environments or application deployment techniques at large scale, which is also not possible on small local testbeds.

This paper is structured as follows: We first describe the MockFog design and explain how it is used within a fog application engineering process (sec. II). Then, we present our proof-of-concept implementation (sec. III) and its evaluation based on microbenchmarks and experiments with an example fog application (sec. IV). Finally, we compare MockFog to related work (sec. V) before a discussion (sec. VI) and conclusion (sec. VII).

[1]https://docker.com

[2]When the application deployment itself is not dockerized, developers need to account for dockerization impacts in the configuration of MockFog [8], [9].

[3]https://aws.amazon.com/ec2/

[4]https://openstack.org

II. MockFog Design

In this section, we give an overview of the MockFog Design. For this purpose, we start with a high-level overview of how MockFog is used within an application engineering process (sec. II-A), describe the abstractions used in our model of a fog infrastructure (sec. II-B), and give an overview of the core components of MockFog and how they instantiate the infrastructure model in the cloud (sec. II-C). Finally, we describe how MockFog can be used in failure testing (sec. II-D).

A. Using MockFog in Application Engineering

A typical application engineering process starts with requirements elicitation, followed by design, implementation, testing, and finally maintenance. In agile, continuous integration and DevOps processes, these steps are executed in short development cycles, often even in parallel – with MockFog, we primarily target the testing phase. Within the testing phase, a variety of tests could be run, e.g., unit tests, integration tests, system tests, or acceptance tests [10] but also benchmarks to better understand system quality levels of an application, e.g., performance, fault-tolerance, data consistency [11]. Out of these tests, unit tests tend to evaluate small isolated features only and acceptance tests are usually run on the production infrastructure; often, involving a gradual roll-out process with canary testing, A/B testing, and similar approaches, e.g., [12]. For integration and system tests as well as benchmarking, however, a dedicated test infrastructure is required. Therefore, we target these kind of tests with MockFog.

For using MockFog, developers first need to model the properties of their desired (emulated) fog infrastructure, namely the number and kinds of machines but also the properties of their interconnections (we describe this in detail in sec. II-B). While this modeling phase is a lot of effort, it only needs to be completed once as the model itself can be persisted and reused. Part of MockFog is a visual editor which supports the modeling phase. Once an infrastructure model has been created, MockFog can automatically instantiate the described infrastructure in the cloud (infrastructure bootstrapping phase) so that developers can deploy a to be tested application version in the emulated fog environment. Finally, the desired tests and benchmarks can be executed – here, MockFog offers capabilities to inject a variety of failures at runtime so that developers can also analyze the fault-tolerance of their application.

We imagine that developers specify the infrastructure model as part of the setup of their deployment pipeline (see figure 1). When a new version of the application has then passed all unit tests, the infrastructure bootstrapping and application deployment phase can be executed as if testing a standard cloud application. Finally, the emulated infrastructure can either be destroyed automatically or the developer can keep it running to interactively play out failure scenarios (see sec. II-D).

B. Fog Infrastructure Model

A typical fog infrastructure comprises edge machines, cloud machines, and possibly also machines within the network

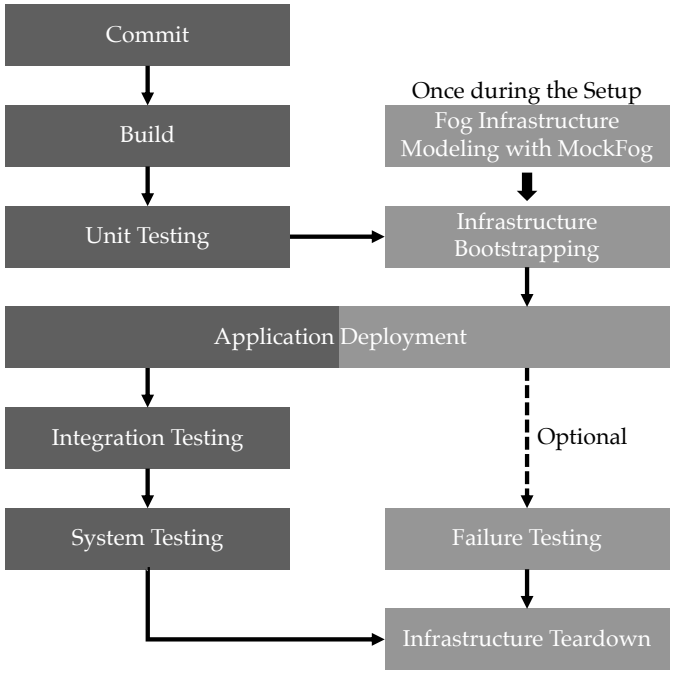

Figure 1. MockFog as Part of an Application's Deployment Pipeline

between edge and cloud [1]. On an abstract level, this can be described as a graph comprising machines as vertices and the networks between machines as edges [13]. In this graph, machines and network connections can also have properties such as the compute power of a machine or the available bandwidth of a connection. During the infrastructure modeling phase, the developer starts with the specification of such an abstract graph before assigning properties to vertices and edges. In the following, we will describe the properties supported by MockFog.

1) Machine Properties: Machines are the parts of the infrastructure on which application code is executed. Fog machines can appear in various different ways, from small edge devices such as Raspberry Pis[5], over machines within a server rack, e.g., as part of a Cloudlet [2], [14], to virtual machines provisioned through a public cloud service such as Amazon EC2.

Table I
Properties of Emulated Machines

Property	Description
Compute Power	Indicator for the Performance
Memory	Amount of Available Memory
Storage	Amount of Available Storage

To emulate this variety of machines in the cloud, their properties need to be described precisely. Typical properties of machines are compute power, memory, and storage – see also table I. Network I/O would be another standard property,

[5]https://raspberrypi.org

however, we chose to model this only as part of the networks in between machines.

While the memory and storage property are self-explanatory, we would like to emphasize that there are different approaches for the measurement of compute power. Amazon EC2, for instance, uses the amount of vCPUs to indicate the compute power of a given machine. This, or the number of cores, is a very rough approximation that, however, suffices for many use cases as typical application deployments rarely achieve 100% CPU load. As an alternative, it is also possible to use more generic performance indicators such as instructions per seconds (IPS) or floating point operations per second (FLOPS). Our current proof-of-concept prototype relies on Docker's resource limits or even the bare number of CPU cores on a cloud VM. If more fine-grained settings are necessary, e.g., based on IPS, an alternative implementation might use tools such as Cpulimit[6] to limit the resources available to a specific application.

2) Network Properties: Within the infrastructure graph, networks connect machines: only connected machines can communicate. In real deployments, these connections usually have diverse network characteristics [4], e.g., slow and unreliable connections at the edge and fast and reliable connections near the cloud, which strongly affects applications running on top of them. These characteristics, therefore, also need to be modeled – see table II for an overview of our model properties. For example, if a connection between machines A and B has a delay of 10ms, a dispersion of 2ms, and a package loss probability of 5%, a package sent from A to B would have a mean latency of 10ms, a standard deviation of 2ms, and a 5% probability of not arriving at all.

Table II
PROPERTIES OF EMULATED NETWORK CONNECTIONS

Property	Description
In-rate	Incoming Bandwidth[7]
Out-rate	Outgoing Bandwidth
Delay	Latency of Outgoing Packages
Dispersion	Delay Dispersion (+/-)
Package loss	Probability of Package Loss
Corruption	Probability of Package Corruption
Reorder	Probability of Package Reordering
Duplicate	Probability of Package Duplication

In most scenarios, machines are not connected directly to each other. Instead, machines are connected to switches and routers which are then connected to each other. We decided to also reflect such virtual routers in our infrastructure model as it reduces complexity of the infrastructure graph by dimensions. See figure 2 for an example with routers and imagine having to model the cartesian product of machines instead. In the graph, network latencies are calculated as the weighted shortest path between two machines. For instance, if the connection between M2 and R1 (in short: M2-R1) has a delay of 5ms, R1-R2 has 4ms, and R2-M6 has 1ms, the overall messaging delay is 10ms.

[6]https://github.com/opsengine/cpulimit

[7]Internally, we model the network graph as shown, e.g., in fig. 2, as two directed graphs – one for each direction.

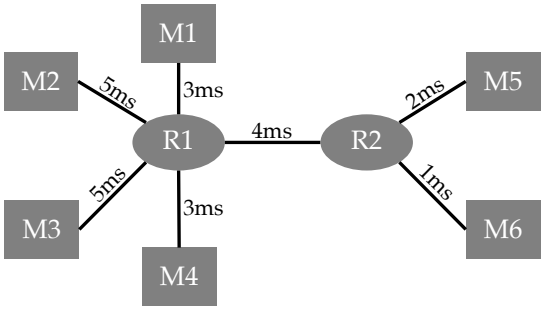

Figure 2. Example: Infrastructure Graph with Machines (M), Routers (R), and Network Latency per Connection

C. Infrastructure Bootstrapping

During the infrastructure bootstrapping phase, the existing infrastructure model is instantiated in the cloud. For this purpose, each fog machine in the infrastructure model is mapped to a single cloud VM. VM type selection is straightforward when the cloud service accepts the machine properties as input directly, e.g., on Google Compute Engine. If not, e.g., on Amazon EC2, the mapping needs to select the smallest VM that still fulfills the respective machine properties. If these are unbalanced, e.g., a small amount of memory combined with a very high number of CPU cores, it may be necessary to specify additional constraints on the resources available to application code. In a next step, the network properties need to be emulated.

For the instantiation of infrastructure models, MockFog has two main components: the *node manager* and the *node agents*. There is only a single node manager instance in each MockFog setup. It parses the infrastructure model, connects to the respective cloud provider, sets up VMs and networks, and installs a node agent on each VM. Node agents, in contrast, manipulate their respective VM to show the desired network characteristics to applications.

Figure 3 shows an example with four VMs: one runs the node manager, two the emulated edge machines, and one a single "emulated" cloud machine. Once the node manager has spawned all instances and installed the node agents, it instructs the node agents to manipulate network properties such as the delay for direct communication between the two edge VMs so that it appears as if all network traffic were routed through the cloud VM.

Overall, the node manager is the point of entry for application developers and, thus, is not only able to spawn new VMs and forward configuration details to node agents. It also offers a browser-based graphical user interface so that developers can easily create, edit, and manage infrastructure models but also handle failure testing (see sec. II-D). Beyond the user interface, the node manager also offers its entire functionality

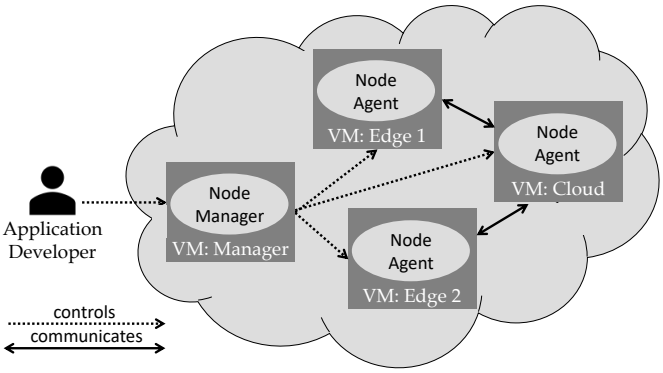

Figure 3. Example: MockFog Node Manager and Node Agents

via API endpoints so that MockFog can easily be integrated into existing automated testing workflows.

While the node agents can also be used to emulate network partitioning, MockFog always asserts that the node manager can continue to reach each VM. For this, we rely on dedicated management networks. We discuss more details of our proof-of-concept implementation of MockFog in sec. III.

Once the infrastructure bootstrapping phase has been completed, the developer can roll-out the application to MockFog in the application deployment phase. For this phase, MockFog provides IP addresses and access credentials for the emulated fog machines that can be used with standard deployment tooling.

D. Failure Testing

Optionally, as also shown in fig. 1, a build process can lead to a failure testing phase. In such a phase, failures are artificially injected to analyze how the application behaves in the presence of such failures, e.g., network partitioning, machine crashes, and others. This is particularly useful, as failures are common in real deployments but will not necessarily occur while the application is being tested. Hence, artificial failures are the go-to approach to test fault-tolerance and resilience of an application [15].

In MockFog, all properties of emulated machines and network connections (tables I and II) can be manipulated at application runtime. For example, it is possible to suddenly limit the amount of available memory (e.g., due to noisy neighbors), make certain connections temporarily unavailable, increase messaging delays or package loss probabilities, or render a machine completely unreachable in which case all communication to and from the respective VM is blocked. As these failures are enacted through the same mechanism as the fog emulation itself, MockFog can store current property settings as snapshots. Developers can then simply preconfigure complex failure scenarios and quickly switch between the scenario snapshots.

For simple scenarios, we envision that all manipulations are done manually in the node manager's user interface, i.e., by updating the infrastructure model. However, to facilitate more complex, automated testing scenarios, all properties can also be updated through the node manager API. Thereby, tools such as Netflix's Chaos Monkey [16] can manipulate properties at random in order to test an application's runtime characteristics in the most extreme situations. The API may also be used in order to integrate MockFog in continuous testing pipelines.

III. Proof-of-Concept Implementation

In this section, we describe the proof-of-concept implementation of MockFog.

MockFog has been developed with the goal of independence from specific IaaS cloud providers and can therefore be extended to the product of choice. Our current proof-of-concept prototype integrates with Amazon EC2 and OpenStack as private cloud. While both versions have been implemented, the one for Amazon EC2 is slightly more powerful and stable than the OpenStack branch, so that we will base the following all on the EC2 branch only.

We have released our proof-of-concept as open source[8] software. In the following, we will describe the three main components *MockFog Initializer*, *Node Manager*, and *Node Agent*.

A. MockFog Initializer

The MockFog Initializer is supposed to run either on the developer's local machine or within the build pipeline. Its main purpose is to start a VM for the Node Manager as well as to deploy and start the Node Manager. This way, a developer only needs to clone our Git repository and run a script instead of managing software stacks manually. At this point the developer grants access to the cloud provider by entering the necessary credentials.

For the implementation of the MockFog Initializer, we rely on the Infrastructure as Code (IaC) paradigm. Following this paradigm, an infrastructure definition tool serves to "define, implement, and update IT infrastructure architecture" [17]. The main advantage of this lies in the ability to define infrastructure declaratively and roll it out in an idempotent way. In our implementation, we use Ansible[9] playbooks combined with Python and Linux shell scripts. Possible extensions of this component would be adding support for other IaaS providers such as the Google Cloud Platform or Microsoft Azure.

B. Node Manager

The Node Manager is the central component serving a RESTful API and a Graphical User Interface (GUI). Based on API or GUI input, it triggers a local IaC component to roll out the Node Agents (IAC in fig. 4), handles import, export, and versioning of infrastructure models, and distributes deduced connectivity metrics to the Node Agents. Through the GUI, application developers can create and edit infrastructure models; the Node Manager performs integrity checks to avoid impossible combinations of machines and their connections while providing feedback to the developer. Furthermore, a versioning feature makes it easy to create model variations (i.e.,

[8] https://github.com/OpenFogStack/MockFog-Meta
[9] ansible.com

Table III
OVERVIEW OF NODE MANAGER SERVICES

Port/Path	Service	Description
80/	nginx	Web server serving the Node Manager's GUI
5001/	flask	Node Manager controller that can reboot single containers
8888/	swagger-ui	Swagger UI that provides API documentation
7474/	neo4j-console	Neo4j Browser for querying, visualization, and manual data interaction
7474/webapi	node-manager-api	API of Node Manager

saving different topologies), which is especially helpful when an application's runtime characteristics should be evaluated for multiple testing infrastructures. The versioning feature also supports import and export of infrastructure models as well as branching of versions, meaning that a slightly different model can be based on a previous version while keeping both. For storage of infrastructure models and versioning, we use a neo4j graph database[10].

In order to mimic changes on an already bootstrapped model, e.g., during failure testing, the Node Manager deduces all necessary properties from the infrastructure model and, subsequently, instructs all Node Agents to override their current configuration in accordance with the updated model. This means that each property change in the infrastructure model causes the Node Manager to call every Node Agent to completely override its current configuration. While this is certainly a significant overhead, we decided on this approach since it reduces the likelihood of consistency issues; should in any way such a problem arise, the Node Manager can simply rebroadcast the model.

In order to be independent from introduced network properties (such as an increased delay) while updating, the Node Manager controls the machines via a dedicated "management network", which is provided by an automatically created router within the MockFog environment and never used by the running fog application. Additionally, we provide a ping endpoint, which allows to keep track of the latency to the machines.

The Node Manager was built with Java EE as enterprise web service framework using JAX-RS, neo4j as graph database system[11], nginx[12] for serving the static frontend parts, Python Flask-Restplus[13] for monitoring tasks, and Swagger[14] as well as the OpenAPI specification[15] for API documentation and testing. All neo4j service classes are built on top of JAX-RS and implement the REST endpoints of the Node Manager within the neo4j server process. Speaking in neo4j terms, the

[10]https://neo4j.com
[11]https://neo4j.com/developer/guide-neo4j-browser/
[12]https://www.nginx.com
[13]https://flask-restplus.readthedocs.io
[14]https://swagger.io
[15]https://www.openapis.org

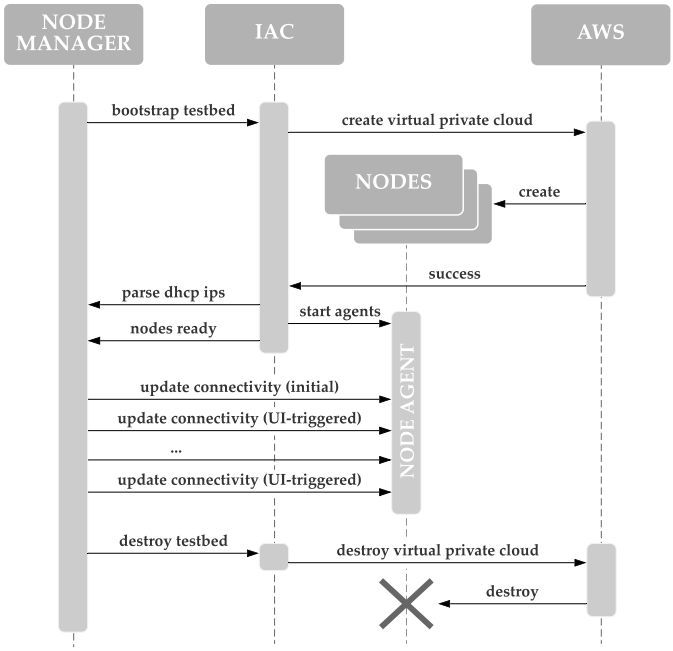

Figure 4. Node Agent Interaction and Lifecycle

Node Manager has been developed as an unmanaged plugin. The subcomponents neo4j, nginx, and swagger are bundled in separate Docker containers for separation of concerns offering the possibility of individual operations management. Table III gives an overview of the services running in the Node Manager.

C. Node Agent

The purpose of the Node Agent is to actually emulate the network properties from the infrastructure model. On each update call, the Node Agent receives an adjacency list containing all other VMs which should be reachable from its corresponding VM. Reachability, here, includes the corresponding specification of its effective metrics, how it should be realized from the viewpoint of the Node Manager's infrastructure model. For implementing the reachability, we use Linux netfilter's CLI iptables; actual metrics (e.g., delay) are controlled via the Linux kernel packet scheduler's CLI NetEm/tc. Generally, Node Agents are configured to drop incoming packages and accept them only if the source appears in the current adjacency list. This allows us to easily emulate network partitioning.

The Node Agent was implemented in Python using Flask[16]. See fig. 4 for an overview of the interaction between Node Manager and Node Agents.

IV. EVALUATION

Beyond our proof-of-concept prototype, we have also evaluated MockFog through experiments: First, we ran a microbenchmark (section IV-A). Second, we deployed an example application on MockFog and a "real" fog testbed and

[16]http://flask.pocoo.org

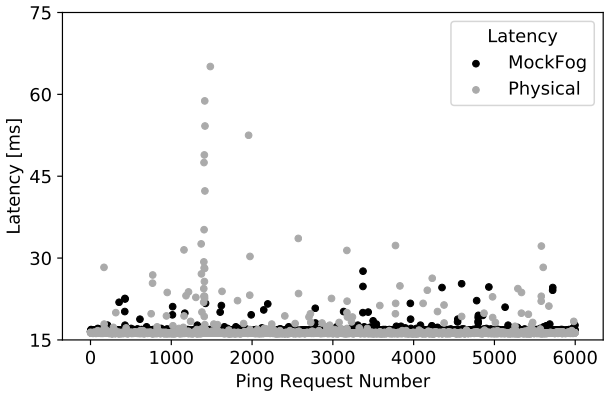

Figure 5. Ping Times of Physical and Emulated Infrastructure

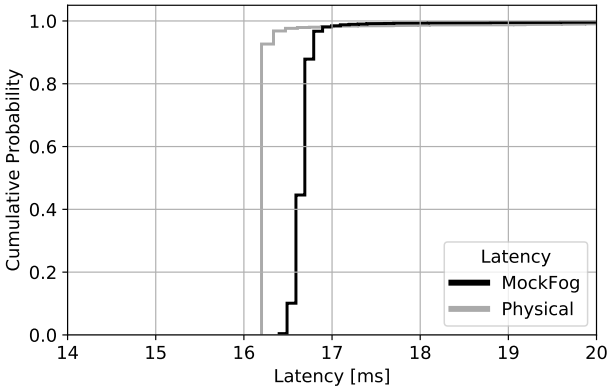

Figure 6. Cumulative Distribution Function of Ping Times

benchmarked the application to compare the application's performance on both infrastructures (section IV-B). In this section, we present the results from both experiments.

A. Microbenchmark

The overall goal of our microbenchmark was to show that it is indeed possible to configure MockFog in a way that applications will have a comparable experience, no matter whether they are running on MockFog or on a real fog testbed. For the testbed, we used an EC2 instance and a Raspberry Pi in our offices. First, we ran a standard ping on the Raspberry Pi to measure the response time from the EC2 instance; the results can be seen in figure 5. Second, we started MockFog and created an infrastructure model of the same test – one emulated edge machine, one cloud machine. We used the measured average response time from the real experiment and provided that as an input parameter for the emulated edge to cloud connection in our MockFog model; we did not make any other changes to the emulated connection. Then, we repeated the same ping test as before, this time running on MockFog.

Figure 5 and 6 show the results of 6000 ping requests each for the connection between the physical Raspberry Pi and the cloud as well as between the emulated MockFog machines. Two ping requests in the MockFog test exceeded 75ms and have been omitted from the figures and we plotted the cumulative distribution function for ping times only for values up to 20ms to improve readability. As we can see, both environments show very similar ping times; MockFog has slightly more outliers which appear to have been caused by network issues on AWS. Overall, this shows that MockFog can provide the behavior described in the infrastructure model even when using only very basic configuration parameters.

We would like to emphasize that this is not an obvious measurement: the packages still need to be sent over the AWS network which does not have zero latency. In fact, MockFog measures real network latencies in the background and tries to account for the measured behavior in the configuration of the node agents. For instance, with a desired emulated network latency of 20ms and a measured real latency of 5ms, MockFog would configure the node agents to cause an artificial delay of only 15ms.

B. Application Benchmark

Applications usually produce more complicated network and processing loads than the ones created by simple ping requests between machines. Therefore, we also implemented a simple prototype of the application scenario introduced by Shneidman et al. [18] and deployed it on both a small infrastructure testbed and on an emulated infrastructure managed my MockFog. We then benchmarked performance parameters within the application.

In the scenario, ambulance cars collect patients and then communicate with all hospitals in a city to find the closest one with available capacity and the required specialists and medical equipment. Once a target hospital has been selected, the ambulance continuously sends medical sensor data to the hospital, e.g., heart rate or arterial oxygen saturation values, so that the emergency room is already fully informed once the patient arrives.

In our implementation, we used RabbitMQ[17] as a pub/sub messaging middleware and built Java components for hospitals, doctors, patients, and ambulance cars that can publish and subscribe to various topics in order to communicate with each other. In our case, RabbitMQ is deployed as a replicated cluster in the hospitals – here represented by a cloud VM each – while the ambulance cars interact with the cloud-based messaging system. For the application benchmark, the components representing doctors and patients only serve as additional load generators for RabbitMQ. For the measurement itself, the ambulance component publishes data packets (10kB) to a particular hospital topic every 250ms. Once the hospital components receives the data packets, it sends back an acknowledgement to the ambulance component. The ambulance component measures this response time as part of our experiment.

For the physical fog infrastructure test, we allocated three EC2 instances in AWS that each represent a single hospital and

[17] rabbitmq.com

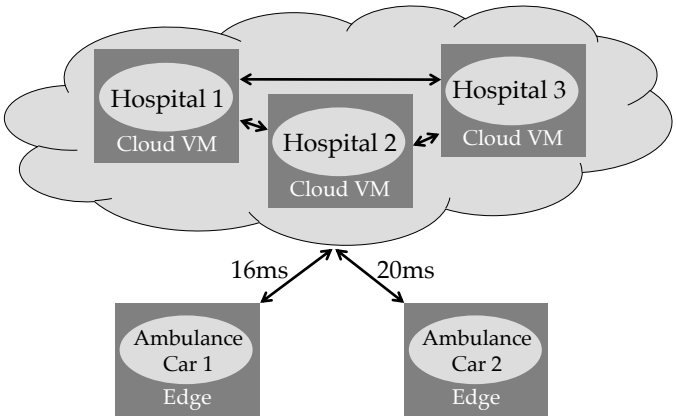

Figure 7. Application Scenario Infrastructure

jointly run a RabbitMQ cluster as well as all non-ambulance code. We also used two Raspberry Pis in our offices connected to the Internet via WiFi, each running a single ambulance component. Ideally, we would have used more than two Raspberry Pis, but we also faced the issue highlighted in the introduction: physical infrastructure is either not yet available for testing, or already in use. This physical setup, including observable average ping times is shown in figure 7.

For the emulated infrastructure, we modeled the described scenario in MockFog and also configured the ping times of 16ms and 20ms respectively between ambulance car and RabbitMQ cluster as shown in figure 7; the experiment was, thus, run on five EC2 instances.

Our results are plotted as histograms in figure 8. In neither of our two experiments, we received messages with a response time below 16ms; all messages with a response time larger than 50ms are placed inside the last bucket of the chart. On the physical infrastructure, most of the delays are between 16ms and 20ms for ambulance car 1, and 20ms and 24ms for ambulance car 2. Both cars, however, experience higher response response times for some of their messages (34.70% and 35.97% of all messages for ambulance cars 1 and 2 respectively). On the emulated infrastructure, the majority of response times for ambulance car 1 is again between 16ms and 20ms, and between 20ms and 24ms for ambulance car 2. However, the amount of messages that experience a higher response time is substantially smaller than on the physical infrastructure (12.84% and 10,45% of all messages for ambulance cars 1 and 2 respectively).

Even though the distribution of delays is different, both ambulance cars have very similar median values on both infrastructures: 18.79ms and 23.08ms on the physical infrastructure, and 18.35ms and 22.35ms on the emulated infrastructure. In fact, only the "long tail" is a bit longer on the physical infrastructure. While the response time is on average (and also based on the median value) very similar, the distribution is not completely identical: Although we can add a delay dispersion with MockFog, the applied dispersion resembles a normal distribution (as this is what NetEm/tc implements),

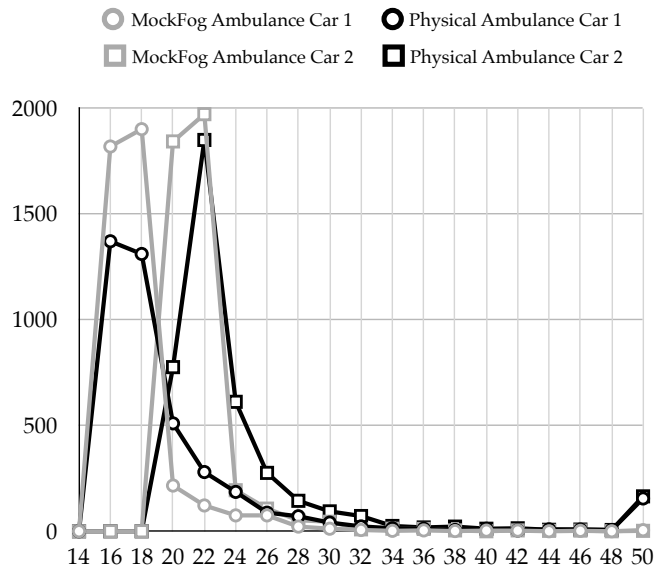

Figure 8. Histogram: Response Times of Ambulance to Hospital Requests

while a geometric distribution appears to be more fitting. This, however, is an effect of the technology used in our proof-of-concept and does not limit our general approach.

In our experiment, we did not measure failure rates (e.g., package loss) or even simply delay dispersion in the physical testbed. This could easily be changed if desired. However, even without this, we can conclude that MockFog provides an infrastructure experience to applications that is sufficiently close to that of their physical archetypes.

V. Related Work

Evaluating and testing distributed applications in fog computing environments can be very expensive as the provisioning and management of needed hardware is costly. Thus, in recent years, a number of approaches have been proposed which aim to enable testing of distributed applications or services without the need for access to fog devices, especially edge devices.

Gupta et al. [7] presented iFogSim, a toolkit to evaluate placement strategies for independent application services on machines distributed across the fog. In contrast to our solution, iFogSim uses simulation to predict system behavior and, thus, to identify good placement decisions. While this is useful in early development stages, their approach can not be used for testing of real application components which we support with MockFog.

Brambilla et al. [5] proposed another system which simulates complex IoT systems with thousands of IoT devices. Moreover, network delays and failure rates can be defined to model a realistic, geo-distributed system. In contrast to MockFog, it does not allow to test existing application components as the approach relies on simulation.

Other simulation approaches include [6] which aims to find good fog application designs or Cisco's PacketTracer which simulates complex networks – both cannot be used for testing of real application components either.

Besides solutions that rely on simulations, a number of researchers focus on the emulation of infrastructure which makes it possible to deploy a real application rather than simulating its workload.

D-Cloud [19], [20] is a software testing framework that uses virtual machines in the cloud for failure testing of distributed systems. However, D-Cloud is not suited for the evaluation of fog applications as users cannot control network properties such as the latency between two machines.

Building on the network emulators MiniNet [21] and MaxiNet [21], Coutinho et al. [22], Mayer et al. [23], and Peuster et al. [24] all target a similar use case as MockFog. Their focus, however, is not on application testing but rather on networks design (e.g., network function virtualization). Based on the papers, the prototypes also appear to be designed for single machine emulations – which limits scalability – while MockFog was specifically designed for distributed deployment. Finally, neither of these approaches appears to support failure testing.

For failure testing, Netflix has released Chaos Monkey [16] as open source[18]. Chaos Monkey randomly terminates virtual machines and containers running in the cloud. The intuition behind this approach is that failures will occur much more frequently so that engineers are encouraged to aim for resilience. Chaos Monkey does not provide the runtime infrastructure as we do, but it would very well complement our approach. For instance, Chaos Monkey could be used to control the emulation of failures in MockFog.

VI. Discussion

While MockFog allows application developers to overcome the challenge that a fog computing testing infrastructure either does not exist yet or is already used in production, it has some limitations. For example, it does not work when a specific local hardware is required, e.g., when the use of a particular crypto chip is deeply embedded in the application source code. Thus, it tends to work well for larger edge machines at least as big as a Raspberry Pis but has problems when smaller devices are involved as they cannot be accurately emulated. The same limitation holds for IoT sensors and actuators, we are still working on mocking them in software in a generic way.

Furthermore, if the communication of a fog application is not based on ethernet or WiFi, e.g., because sensors communicate via a LoRaWAN [25] such as TheThingsNetwork[19], MockFog's approach of emulating connections between devices does not work out of the box as these sensors expect to have access to a LoRa sender. With additional effort, however, application developers could adapt their sensor software to use ethernet or WiFi when no Lora sender is available.

Also, emulating real physical connections is difficult as their characteristics are often influenced by external factors such as other users, electrical interferences, or natural disasters. While it would be possible to add a machine learning component to MockFog that updates connection properties based on past data collected on a reference physical infrastructure, it is hard to justify this effort for most use cases.

Finally, our prototype currently starts a VM for every single node. This does not scale well when the infrastructure model comprises hundreds of machines. Here, our approach could easily be adapted to deploy the application code from multiple small edge devices on only a few larger VMs.

[18]https://github.com/Netflix/chaosmonkey

[19]https://www.thethingsnetwork.org

VII. Conclusion

In this paper, we proposed MockFog, a system for the emulation of fog computing infrastructure in arbitrary cloud environments. MockFog aims to simplify the testing of fog applications by providing developers with the means to define and bootstrap an emulated testing infrastructure that can easily be manipulated to facilitate different scenarios or evaluate how failing machines and unavailable connections affect tested applications. We evaluated our approach through a proof-of-concept implementation as well as a number of experiments with an example scenario and microbenchmarks and conclude that MockFog is capable of emulating an infrastructure that is very similar to its physical counterpart and suited for many fog computing cases.

Acknowledgment

We would like to thank Sören Becker, Miro Conzelmann, Michael Narodovitch, Meike Stoldt, and Franz Tscharf who contributed to the implementation of our proof-of-concept prototype within the scope of a master's project.

References

[1] D. Bermbach, F. Pallas, D. G. Pérez, P. Plebani, M. Anderson, R. Kat, and S. Tai, "A research perspective on fog computing," in *2nd Workshop on IoT Systems Provisioning & Management for Context-Aware Smart Cities (ISYCC)*, vol. 10797. Springer, 2018, pp. 198–210.

[2] R. Mahmud, R. Kotagiri, and R. Buyya, "Fog computing: A taxonomy, survey and future directions," in *Internet of Everything: Algorithms, Methodologies, Technologies and Perspectives*. Springer, 2018, pp. 103–130.

[3] M. Grambow, J. Hasenburg, and D. Bermbach, "Public video surveillance: Using the fog to increase privacy," in *Proceedings of the 5th Workshop on Middleware and Applications for the Internet of Things*. ACM, 2018, pp. 11–14.

[4] P. Varshney and Y. Simmhan, "Demystifying fog computing: Characterizing architectures, applications and abstractions," in *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*. IEEE, 2017, pp. 115–124.

[5] G. Brambilla, M. Picone, S. Cirani, M. Amoretti, and F. Zanichelli, "A simulation platform for large-scale internet of things scenarios in urban environments," in *Proceedings of the The First International Conference on IoT in Urban Space*. ICST, 2014.

[6] J. Hasenburg, S. Werner, and D. Bermbach, "Supporting the evaluation of fog-based IoT applications during the design phase," in *Proceedings of the 5th Workshop on Middlware and Applications for the Internet of Things*. ACM, 2018.

[7] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, and R. Buyya, "iFogSim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments," *Software: Practice and Experience*, vol. 47, no. 9, pp. 1275–1296, 2017.

[8] M. Grambow, J. Hasenburg, T. Pfandzelter, and D. Bermbach, "Is it safe to dockerize my database benchmark?" in *Proceedings of ACM SAC, DADS Track*. ACM, 2019.

[9] ——, "Dockerization impacts in database performance benchmarking," in *Technical Report MCC.2018.1*. TU Berlin& ECDF, 2018.

[10] M. Winter, M. Ekssir-Monfared, H. M. Sneed, R. Seidl, and L. Borner, *Der Integrationstest*. Hanser, 2013.

[11] D. Bermbach, E. Wittern, and S. Tai, *Cloud Service Benchmarking: Measuring Quality of Cloud Services from a Client Perspective*. Springer, 2017.

[12] G. Schermann, D. Schöni, P. Leitner, and H. C. Gall, "Bifrost: Supporting continuous deployment with automated enactment of multi-phase live testing strategies," in *Proceedings of the 17th International Middleware Conference on - Middleware '16*. ACM Press, 2016, pp. 1–14.

[13] A. Kajackas and R. Rainys, "Internet infrastructure topology assessment," *Electronics and Electrical Engineering*, vol. 7, no. 103, p. 4, 2010.

[14] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for VM-based cloudlets in mobile computing," *IEEE Pervasive Computing*, p. 9, 2009.

[15] D. Bermbach, L. Zhao, and S. Sakr, "Towards comprehensive measurement of consistency guarantees for cloud-hosted data storage services," in *Performance Characterization and Benchmarking*, R. Nambiar and M. Poess, Eds. Cham: Springer International Publishing, 2014, pp. 32–47.

[16] A. Tseitlin, "The antifragile organization," *Communications of the ACM*, vol. 56, no. 8, pp. 40–44, 2013.

[17] K. Morris, *Infrastructure as Code: Managing Servers in the Cloud*, first edition ed. O'Reilly, 2016.

[18] J. Shneidman, P. Pietzuch, J. Ledlie, M. Roussopoulos, M. Seltzer, and M. Welsh, "Hourglass: An infrastructure for connecting sensor networks and applications," *Harvard Technical Report*, vol. TR-21-04, 2004.

[19] T. Banzai, H. Koizumi, R. Kanbayashi, T. Imada, T. Hanawa, and M. Sato, "D-cloud: Design of a software testing environment for reliable distributed systems using cloud computing technology," in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE, 2010, pp. 631–636.

[20] T. Hanawa, T. Banzai, H. Koizumi, R. Kanbayashi, T. Imada, and M. Sato, "Large-scale software testing environment using cloud computing technology for dependable parallel and distributed systems," in *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 2010, pp. 428–433.

[21] R. L. S. de Oliveira, C. M. Schweitzer, A. A. Shinoda, and Ligia Rodrigues Prete, "Using mininet for emulation and prototyping software-defined networks," in *2014 IEEE Colombian Conference on Communications and Computing (COLCOM)*. IEEE, 2014, pp. 1–6.

[22] A. Coutinho, F. Greve, C. Prazeres, and J. Cardoso, "Fogbed: A rapid-prototyping emulation environment for fog computing," in *Proceedings of the IEEE International Conference on Communications (ICC)*. IEEE, 2018.

[23] R. Mayer, L. Graser, H. Gupta, E. Saurez, and U. Ramachandran, "EmuFog: Extensible and scalable emulation of large-scale fog computing infrastructures," *arXiv:1709.07563 [cs]*, 2017.

[24] M. Peuster, H. Karl, and S. van Rossem, "MeDICINE: Rapid prototyping of production-ready network services in multi-PoP environments," in *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, 2016, pp. 148–153.

[25] J. d. C. Silva, A. M. Alberti, P. Solic, and A. L. L. Aquino, "LoRaWAN - a low power WAN protocol for internet of things: a review and opportunities," in *2017 2nd International Multidisciplinary Conference on Computer and Energy Science (SpliTech)*. IEEE, 2017.