2019 IEEE International Conference on Autonomic Computing (ICAC)

EMU-IoT - A Virtual Internet of Things Lab

Brian Ramprasad
Dept of El. Eng. and Comp. Sci.
York University
brianr@yorku.ca

Marios Fokaefs
Dept of Comp. and Soft. Eng.
Polytechnique Montreal
marios.fokaefs@polymtl.ca

Joydeep Mukherjee
Dept of El. Eng. and Comp. Sci.
York University
jmukherj@yorku.ca

Marin Litoiu
Dept of El. Eng. and Comp. Sci.
York University
mlitoiu@yorku.ca

***Abstract*—Internet-of-Things technologies are rapidly emerging as the cornerstone of modern digital life. IoT is the main driver for the increased "intelligence" in most aspects of everyday life: smart transportation, smart buildings, smart energy, smart health. Nevertheless, further progress and research are in danger of being slowed down. One important reason is the cost of infrastructure at scale. The difficulties in setting up very large IoT networks do not permit us to stress test the systems and argue about their performance and their durability. To tackle this problem, this work proposes EMU-IoT, a virtual lab for IoT technologies. Using virtualization and container technologies, we demonstrate an experimentation infrastructure to enable researchers and other practitioners to conduct large scale experiments and test several quality aspects of IoT systems with minimal requirements in devices and other equipment. In this paper, we show how easy and simple it is to set up experiments with EMU-IoT and we demonstrate the usefulness of EMU-IoT by conducting experiments in our lab.**

***Index Terms*—IoT, networks, cloud computing, containers, performance, empirical software engineering**

I. Introduction

The number of IoT devices has grown exponentially over the past decade. To investigate and learn about the behavior of IoT networks at scale is not economically feasible. Although the cost for hardware devices may be significantly reduced, the acquisition of a large quantity of specialized equipment can still be prohibitive for research. Despite this, researchers have been exploring the IoT domain and have contributed simulation and emulation tools. Nevertheless, exactly because of cost and inefficiency concerns, these efforts are fragmented and they focus only on parts of an IoT network and in some cases only for a single scenario [1]–[4].

The primary challenge with designing these research tools is how to create a platform that can be used to evaluate the performance of a diverse set of applications and devices from end to end and in a uniform way [5]. Without collecting application performance data in a uniform way, the derived metrics may not be comparable because different sampling methods are used. For example, CPU utilization can be polled from the hardware, the operating system, the container level or the application level. This could yield different results depending on which processes are included in the calculation of the CPU usage metric. Also, each application will use memory and CPU in a different way because each application may have different processing patterns. Sometimes these applications may be installed on different hardware where the type of available computing resources affects the performance of the application. To the best of our knowledge there is no existing system that can provide an end-to-end evaluation of an IoT network that can also be installed on the production hardware. Our motivation for this work emerged from our previous research [6], where we needed to be able to run experiments and perform scalability tests for our back-end data infrastructure for a proposed building management system in the absence of the actual IoT infrastructure and configuration. However, we were unable to find a solution that could help us to evaluate the entire IoT system (the devices and the IoT application). Most tools focused on either the IoT devices or the application performance, but not both.

Towards the goal of providing a solution to scale and evaluate IoT networks in a uniform way, we propose a virtual lab, the Emulated Internet-of-Things Lab (EMU-IoT). EMU-IoT is an adaptable architecture and a smart testing framework which network designers can implement and will allow them to reliably scale an IoT network in an autonomic fashion. Autonomic in this context is defined as a system that is capable of continuously monitoring the managed system with the ability to take corrective actions to achieve the performance goals of the application. In this case, a corrective action would be detecting that a target utilization has been surpassed and a scaling action is required to maintain the quality of service on the network. A quality of service goal would be defined by the user. For example, based on the goal of preventing the CPU utilization from going over a certain value, which has been known to affect the response time of the application, a corrective action could be to horizontally scale the infrastructure by adding a new virtual machine thus giving us more CPU cores. Our overall goal is to provide an emulation environment so that others can install our platform and be able to investigate how their IoT application will perform on their specific network.

Towards this goal we make the following contributions:
We demonstrate a customizable virtual lab (EMU-IoT) that can be used to model heterogeneous IoT networks on an adaptable architecture. We are able to deploy all the necessary components to have a fully working solution that is reproducible by others. The software is executable on nearly any cloud computing service and can be installed without the need of any specialized hardware. EMU-IoT in its present state is capable of monitoring any IoT application as long as it is containerized and can easily be adapted to monitor non-containerized applications. **A methodology and specification is created to define what an emulated IoT device is.** We provide a generic design and a minimum set of characteristics that an emulated IoT device must have. This allows EMU-

2474-0756/19/$31.00 ©2019 IEEE
DOI 10.1109/ICAC.2019.00019

IoT to be extended so that others can create any software defined version of a physical IoT device to meet their custom requirements.

II. Challenges with IoT Deployments

In this section we describe some of the challenges with running IoT experiments. Specifically, we focus on the challenges that consider the high cost aspects of building and maintaining physical labs, experimental complexity and future flexibility in building IoT networks.

In a typical IoT deployment, we have IoT devices, an aggregation point to receive incoming data from these devices, and various applications that use this data for different purposes. For example, in the case of smart homes and buildings we can have many embedded environmental sensors, where we need both real time information and the ability to perform historical analysis on data streams. As shown in Figure 1, we can have homes distributed across different cities that may send their data to a single gateway. Message brokers receive the data from the gateways where they can later be retrieved by a stream processing application for analysis. These stream processing applications may then store the results of the analysis in a persistent database.

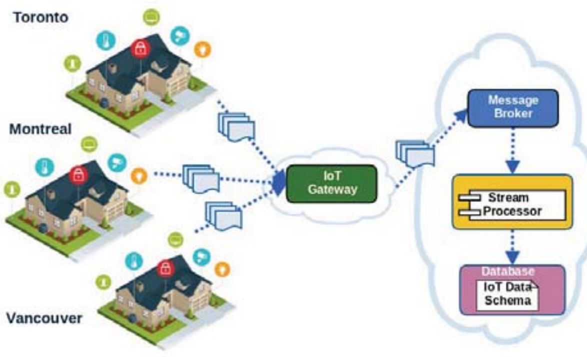


Figure 1: Example Smart Home Streaming Application

A smart home in North America is predicted to have a large number of sensors and that number dramatically increases to over 100 per home where assisted living technologies are installed [7]. We can imagine that data streams from hundreds or even thousands of homes would require extensive infrastructure to support such applications. Homes are naturally geographically dispersed and some maybe in remote areas away from the data centers that process the data streams. Using the example of smart homes, we argue that to accurately plan for future capacity and to properly stress test IoT applications with realistic IoT workloads, the emulation of IoT devices and IoT networks alone is not enough. We must be able to deploy the actual data aggregation and stream processing applications along with the emulated IoT devices. Deploying the applications and virtual devices together is similar to the way they would be deployed in a production environment. This gives us a more realistic picture of how an application will behave. Keeping this in mind, we now discuss the challenges associated with such deployments.

A. Physical Challenges

IoT devices exist as physical devices in the world today. Therefore, to execute an experiment involving physical sensors, one must be able to procure devices, which can present a challenge primarily from a cost perspective. While some sensors may be relatively inexpensive to acquire, other devices maybe be very expensive which can lead to a reduction in the number of actual sensors used in physical lab experiments [8]. Physical sensors require human labour costs and supporting infrastructure. For example, deployment costs can range from $2.50/sqft which would be approximately $250,000 in a 100,000 sqft building. This is a large investment before knowing how an IoT application will perform [9].

B. Experimental Challenges

In this section we discuss some of the challenges faced by a virtual lab architecture in order to emulate real world IoT networks. The aim of this work is to be able to emulate the topology and behavioral patterns that may occur in an IoT network.

1) *Geographical Distribution*
   The very nature of IoT devices is that they are dispersed geographically meaning that they are externally located and away from a centralized computing infrastructure. For example, one may have a centralized analytics application located in one city, but IoT devices may spread across many cities and countries. This is done to reduce the cost and complexity of the IoT network. Therefore, to replicate this scenario and get results similar to a production deployment, we need an emulator that is capable of allowing users to install their particular IoT application into their physical production hardware rather than in a test environment where the generated results could be different.
2) *Temporal Distribution*
   IoT devices are typically configured to suit the usage patterns of the environment in which they are deployed. For example, within a work environment, during the day a temperature sensor may be configured to take more frequent readings because during these hours more humans can imply greater temperature fluctuations in the space. Since we have more frequent readings, the load on the network becomes larger. Conversely at night, the temperature is much more stable because everyone has gone home and now we can configure the sensor to take readings every hour instead of every 5 minutes during work hours. In this scenario, an emulator for IoT networks should be parameterizable to include this temporal feature to replicate this scenario.
3) *Heterogeneity*
   Perhaps the most obvious characteristic of IoT networks is the variety of device types that exist in the environment. Many new devices are being deployed and others are

being decommissioned on a regular basis. This presents two main challenges. First, we need an emulator that is easily extendable and generic enough so that someone can create new emulated IoT devices. Second, we need to design the emulator in such a way that it does not disturb the stability of the existing network when we remove and add new devices. This is a critical feature for the emulator, because for real world networks, downtime is not acceptable.

4) *Network Connectivity Types*
As previously mentioned, there could be many types of sensors on an IoT network. These devices may also use different technologies to establish connections to the cloud network, for example, Bluetooth, Wi-Fi,and Ethernet. These different types of connectivity have different bandwidth speeds. An IoT emulator should have this feature so that we can compare the delay and bottlenecks between using different network connectivity types.

5) *Network Protocol Variety*
IoT networks can also be diverse in terms of the protocols that are used to provide communication between devices and the computing infrastructure. Additionally, existing protocols are always being updated and new ones may appear in the future. An emulator should be designed in such a way that you can use any communication protocol. No reconfiguration should be necessary to implement a new protocol or to have multiple protocols running at the same time.

6) *Infinitely Scalable Design*
A common theme appearing in the research of IoT networks is that the number of connected devices is growing exponentially and this trend is expected to continue for the foreseeable future. A major concern for application developers and network designers is how to prepare for the growth ahead. There are two major concerns in this problem space: a) how to quickly scale up to meet the resource demands incurred by an exponential growth of devices, and b) how to avoid having idle resources in our infrastructure that will unnecessarily increase cost. An emulator must have the capability to deal with these two problems.

7) *Disaster Handling*
An often overlooked scenario when designing an IoT network is the ability to handle the failure of application components and the IoT devices themselves. The failure of IoT devices is a common occurrence since devices operate in the external environment and are subjected to harsh outdoor conditions. In the physical world, the IoT operator will have to directly interact with these devices and either cut their power off or disconnect them from the network. This may be difficult if the device is in a remote location that is hard to access. An emulation environment should allow users to easily and quickly execute disaster scenarios to learn about the effects on the IoT network and applications that depend on these devices.

8) *Security Threat Handling*
As per our smart home example these devices exist in an uncontrolled environment. They are subject to attacks and may become compromised. Applications that process streaming data need to know how to deal with rogue or compromised devices in their network and how to deploy a threat mitigation response. With emulated IoT devices the device behaviour can be customized to evaluate a wide variety of scenarios to perform vulnerability testing on production systems.

9) *Cyber Physical and Virtual Device Coexistence*
Very often it is the case that there are existing IoT devices deployed in production and it is possible that the addition of new devices may put stress on an application that makes use of the IoT data. In order to evaluate the computing resources needed to support additional devices in a production environment, we can use emulation to inject software defined versions of the IoT devices that will also send data to the production application. From the application perspective, it should be opaque that the device that is sending its data is emulated and not physical. This approach gives us great flexibility with respect to testing before laying out any capital costs and does not disturb any running application in production.

III. EMU-IoT Components and Architecture

In this section we discuss the major components of EMU-IoT and how they inter-operate to provide an end-to-end solution for evaluating IoT networks and application performance. Our approach is based on a microservices architecture using containers, which allows for rapid instantiation, customizability and portability of the components. Using microservices also allows EMU-IoT to provide robust orchestration capabilities to execute large scale experiments and to collect performance metrics in a uniform way across the entire deployment. Our current implementation is based on Docker but any other container provider can be used.

A. Device Properties

A virtualized IoT device must fully emulate the behavior and characteristics of the actual device. In the context of IoT emulation, we will emulate the three generic properties that a virtualized IoT device should have: connectable, configurable, and deployable. Our goal is to provide a generic reference architecture that can be extended to suit the needs of a particular deployment.

1) *Connectable*
IoT devices vary in terms of their ability to connect to a receiving device that will read the emitted data. The typical connection types are over Bluetooth, WIFI, and hardwired (serial, Ethernet). A receiving device could be a controller that aggregates several sensors or other IoT devices that communicate in a peer-to-peer topology. The consequence of this is that different connectivity modes may have different transmission rates. For example, transmitting data over Ethernet is faster compared to WIFI and Bluetooth. To deal with this variation, a virtualized

IoT device must have the ability to set its transmission type so you can model the actual physical device. When instantiating an IoT device in the EMU-IoT environment, the connectivity type can be set at run time.

2) *Configurable*
Since IoT devices also vary greatly in terms of the types of onboard sensors (temperature, movement, video, audio, etc.), developing a software-based component that is configurable is more practical compared to several individual devices. In the first iteration of our design we focused on creating a virtualized IoT device with the ability to configure the emission rates to simulate different connectivity types and setting the reading ranges for a temperature and luminosity sensor. In future iterations, we hope to expand the number of generic sensor types that are supported.

3) *Deployable*
The most important feature of the EMU-IoT platform is the ability to rapidly instantiate IoT devices. To achieve these goals, we use a microservices approach by containerizing the software component so that it can be quickly activated. A virtualized sensor is a small lightweight application that begins emulating the behavior of an IoT device once a container has been started. The primary advantage of this is that an unlimited number of virtualized IoT devices can be created and they can be selectively dispersed on different networks according to the needs of the user. Another advantage of using microservices is that you can have a common way to communicate with the containers even though different IoT software components might be running inside the container.

B. Virtualized IoT

In this section we describe the two main virtualized components in EMU-IoT, the *Device* and the *Gateway*, which enable workloads to be executed. These components can be rapidly instantiated and deployed at scale using a microservices approach.

1) *Device*
A virtualized IoT *Device* in the context of the EMU-IoT lab is an encapsulated service that emits data. It adheres to the three properties as described in Section III-A which are connectability, configurability, and deployability. As shown in Figure 2, we have three different components inside a virtualized IoT device. The first component is the *Configuration Interpreter*, which allows users to modify the behavior of the device at build time, where the parameters are passed to the service running inside the container. Once the configuration has been set, the second component, the *Data Generator* service begins to produce the emulated data. This data is then encoded in the appropriate format for the transport protocol being used, and then the third component, the *Data Emitter*, transmits the information to a service that is external to the virtualized IoT device. A virtual IoT device can also be configured to accept new data at runtime if the device needs to obtain new information once the data generation service has been started.

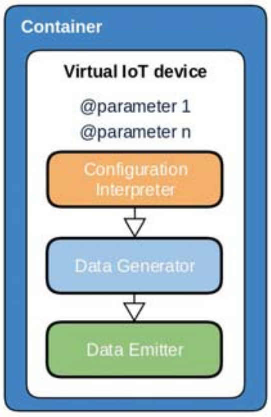


Figure 2: Virtualized Device

2) *Gateway*
A virtualized IoT *Gateway* in the context of the EMU-IoT lab is an encapsulated service that receives, formats, and forwards data onward to an external service that collects data from many gateways. Virtual *gateways* may also support physical IoT Devices.

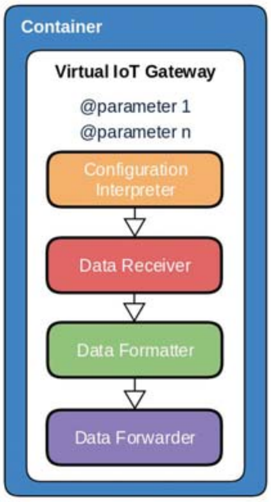


Figure 3: Virtualized Gateway

As shown in Figure 3, we have four different components inside of a virtualized IoT gateway. We have a *Configuration Interpreter*, which accepts parameters that are passed to the service inside the container to modify the behavior of the gateway. Once the configuration has been set, the *Data Receiver* service starts and waits for incoming data from the physical or virtualized IoT devices. This data is then checked to make sure it is in the correct format by the *Data Formatter* and then each reading from the IoT device is formatted into a common message standard that is used on the IoT network. The *Data Forwarder* then makes a connection to a data aggregation service that is waiting for this information.

C. Application Architecture in EMU-IoT

A typical IoT application consists of IoT devices, the gateways that provide connectivity and the back-end applications that consume the IoT data. In this section, we discuss the underlying architecture in EMU-IoT that supports a running IoT application. We also describe the pipelines that allow the movement of data between the components in the IoT network.

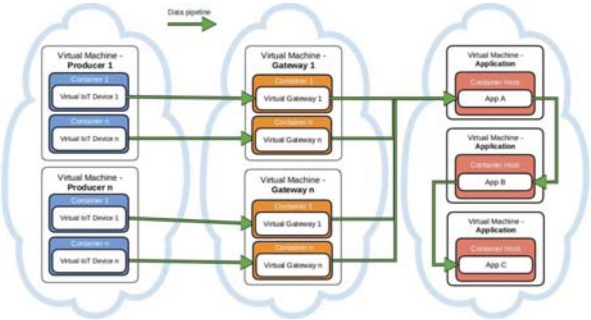


Figure 4: Network Architecture

The architecture for the IoT network is shown in Figure 4. As seen in the figure, the IoT network has 3 separate components that can operate independently of each other. These 3 components are the *IoT producers*, the *IoT gateways* and the *IoT applications*. The producers emulate IoT devices and stream the IoT data to the gateways which in turn forward the data to the applications. In this particular example as shown in Figure 4, once the IoT data is forwarded to the application component, the data is passed between various applications, such as from App A to App B and then to App C. We note that there can be fewer or more than 3 applications in this component and that these applications can be standalone services that are not dependent on each other. As seen in Figure 4, all 3 components run on a containerized environment. The use of containers is a mandatory requirement for the producers and the gateways, but the applications can run directly on the host. Next, we discuss these 3 components in detail.

D. Producer Host

The producer host consists of virtualized IoT devices that *produce* data which is emitted and read by an external service. As shown in Figure 5 we have a Virtual Machine (VM) that can host many instances of the virtual IoT device. The purpose of this design is to allow us to deploy multiple IoT devices easily on the producer VM as needed. Each virtual IoT device is deployed as a single container, as shown in Figure 5. The choice of deploying the devices on containers is dictated by the fact that containers are designed to be lightweight, thus allowing a large number of virtual IoT devices to be run on a single producer VM. This design overcomes the challenges described in Section II by providing a way to deploy sensors without incurring large costs and being able to create and remove them on demand which is difficult with physical devices. We note that the virtualized IoT devices are created one per container as this allows us to better emulate a standalone IoT physical device. In contrast, creating multiple virtualized devices as threads sharing resources inside a single container does not properly emulate a realistic IoT deployment scenario.

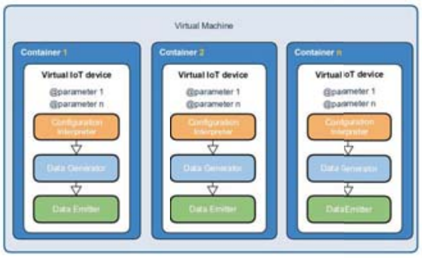


Figure 5: Producer Host

E. Gateway Host

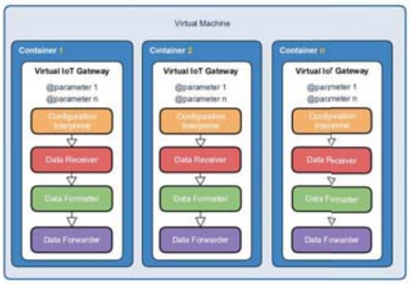


Figure 6: Gateway Host

The gateway component consists of a VM that hosts the virtual IoT gateways where the virtualized IoT devices from the producer VM transmit their data to. The gateway component is shown in detail in Figure 6. This figure illustrates that the gateway VM hosts multiple containers and a single instance of a virtual IoT gateway is run inside each container. Each virtual IoT gateway is configured to simultaneously receive data from multiple IoT devices. Hence, we do not need as many virtual IoT gateways as the number of virtual IoT devices. This is a key feature of EMU-IoT as we can experiment with different workloads on the gateways to determine the optimal number of gateways needed to support a set of IoT devices. This is important because the computing resources needed for a virtual IoT gateway are much higher compared to a virtual IoT device, and therefore optimizing our computing resources by limiting the number of virtual IoT gateways is crucial. We note that similar to the producer host, the gateway host can have more than one container with a IoT device or virtual gateway.

F. Application Component

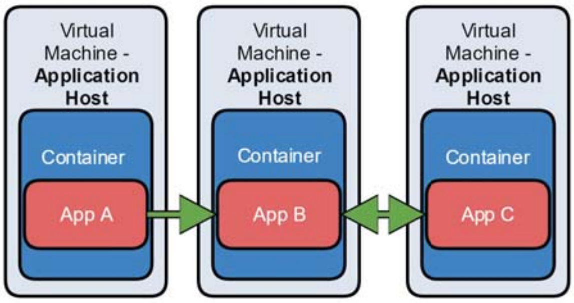

Virtual Machine - **Application Host**
Container
App A
Virtual Machine - **Application Host**
Container
App B
Virtual Machine - **Application Host**
Container
App C

Figure 7: Application Component

The application component is shown in Figure 7. As seen in the figure, the application component consists of one or more VMs that hosts containerized applications which support the IoT services running on the network. As mentioned before, these applications can also be run directly on the VMs without the need for containers, or in some cases, even directly on the hardware. The applications are typically services that ingest, process and store the incoming data from the various IoT devices on the IoT network. Typically, we are interested in monitoring the resource usage patterns of the IoT applications in order to maintain the quality of service for these applications on the IoT network.

G. EMU-IoT Simulator

In this section we briefly discuss the main components of the EMU-IoT platform as shown in Figure 8. Together, this collection of services provides users with the ability to define and execute a wide variety of IoT experiments.

1) *IoT Orchestrator*
The IoT Orchestrator is the main driver of the EMU-IoT platform. The IoT Orchestrator defines the configuration parameters of the experiment, such as the experiment type (bottleneck detection, resource prediction, network latency, etc), experiment duration, and the particular resource metrics such as (CPU,DISK, MEMORY, etc) that are to be observed and recorded. At this moment, the IoT Orchestrator is script-based but we hope to provide a Web based user interface in the future.
2) *IoT Monitor*
An IoT Monitor is responsible for observing and collecting the resource metrics for a single application. Using this approach allows us to have a uniform way of collecting resource usage metrics from the applications that support the IoT devices. In the IoT Monitor class, the data collection functions can be overwritten to extract data from another source other than Docker. This makes it quite easy to pull data from any source in case a commercial monitor such as `Prometheus`[1] is being used.
3) *IoT Monitor Manager*
The IoT Monitor Manager is responsible for coordinating the metrics collection process from the individual IoT Monitors. Our example implementation is based on Docker, which exposes a metrics API to obtain information about the resource consumption of the containers. The manager aggregates the information from each IoT Monitor which observes individual application instances. IoT Monitor Manager is platform agnostic, which means that it can manage monitors that pull data from sources other than Docker as long as the monitor adheres to the generic interface we have defined.
4) *IoT Experiment*
The IoT Experiment component drives the actions that the user is trying to enact in order to see observable changes in the application performance. The actions are based on the type of experiment that needs to be run. The user can provide the addresses of the Docker containers and the applications that will be monitored. IoT Experiment then implements the Smart Testing Framework.
5) *Smart Testing Framework*
The framework is based on the concept of generating test cases in order to drive IoT experiments or to autonomically trigger actions to scale the IoT network up and down in a production setting. In a scenario where additional IoT devices are being deployed it is important to know what computing resources are needed to support the applications that will process the additional incoming data. A prediction model would be used to estimate the amount of resources required. The Smart Testing Framework provides a facility for a user to implement a custom prediction model. More details about the framework can be found here [10]. For the experiments described in this paper a simple linear regression module was used from the `Python Statsmodels API`[2].
6) *IoT Experiment Linear Regression and ML*
These components are examples of user provided libraries that are fed data from the IoT Monitors and then used by the Smart Testing Framework to generate test cases for predictive modeling. Depending on the type of data that is being generated by the IoT devices, different libraries can be substituted.
7) *IoT Device Service*
The IoT Device Service provides the simulator with the ability to instantiate software defined versions of IoT devices. As described in section III-A, these are containerized virtual devices and this service allows you to start a container that encapsulates the source code for the emulated IoT device. The service provides a set of generic interfaces that apply to all IoT device types. Currently we have implemented a temperature and light device in addition to an IoT camera as shown in figure 8. As long as a IoT device can be created inside a Docker image, this service can be used to turn on and off the

[1] https://prometheus.io/

[2] https://www.statsmodels.org/stable/index.html

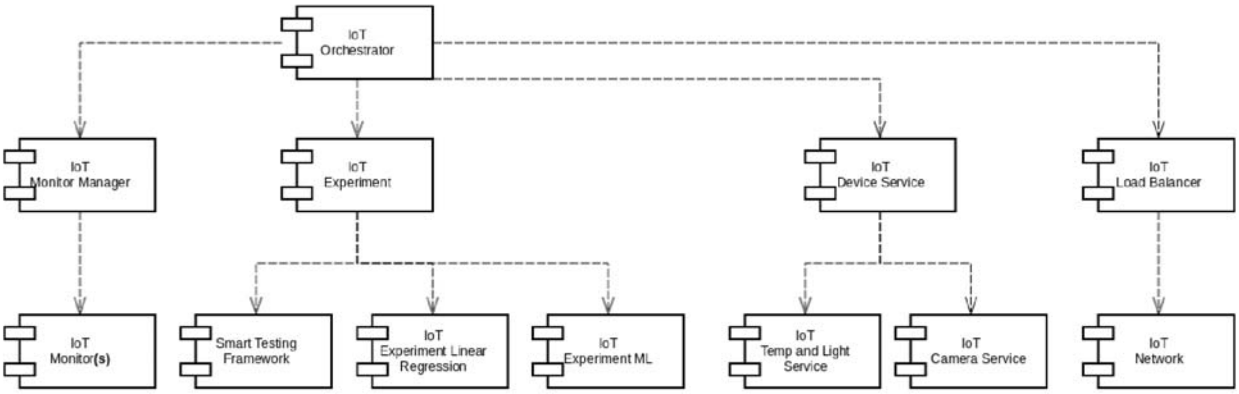


Figure 8: EMU-IoT Simulator

virtualized device. As shown, in Figure 8 the service can be extended for various device types. It is also possible to combine multiple sensors inside a single container if required. For example, to emulate an entire smart home that emits aggregated data, then the application should be designed to combine this data into a single tuple (a row of IoT data) for emission.

8) *IoT Load Balancer*
The IoT Load Balancer maintains information about the resources on the IoT network and is used by the simulator when deciding where best to place virtualized IoT devices and gateways. The available resources determine the host's ability to handle virtual devices. For example, we can logically define how many devices a given physical host can support based on the type. A temperature sensor is computationally light, so we can have large numbers of them on a given host, whereas a camera device is much more computationally intensive, so a host can only have a few of them running. When a request arrives to create a device, the load balancer will decide where to place it. This can be done by defining a new `distribution policy` method. A policy can be used to place virtual devices on specific hosts based on geographic location or a round robin policy. Our example application uses a fill first policy which loads the maximum number of virtual devices on a host then moves to the next available host.

9) *IoT Network*
The IoT Network component maintains information about the physical layout of the entire IoT network and the interconnected links between the different hosts. It provides look up methods for the simulator as shown in Figure 4. For example, if we want to find out which hosts can serve virtualized IoT devices or serve as hosts for our big data application, this service will provide us with this information.

IV. Experiment Setup and System Configuration

In this section, we describe how EMU-IoT can enable users to run experiments using virtual components that would otherwise be cost prohibitive if physical devices were used. We provide details on how to configure the parameters to suit a particular IoT application deployment and we also discuss the variety of experiments that are possible. In section V we discuss the results of experimenting with an example IoT application. The EMU-IoT platform can be cloned from the Github repository [3].

A. Defining an Experiment

The first step in setting up an experiment or ongoing testing of a live system is to define the goal of the experiments. In our example scenario we have an IoT application that collects data from environmental sensors in real time, analyses the data, and persists the information to storage. Therefore the goal for this application is to have fast responses to user queries over those data streams. To achieve this goal, it is a strict requirement that the computing resources (CPU, Disk, Memory) do not become saturated such that they negatively impact response time. We discuss in the following sections how to setup the simulator to evaluate this scenario using EMU-IoT.

B. Environment Prerequisites

EMU-IoT is a containerized platform that is `Docker`[4] compatible. Docker is required to be installed with the remote management API enabled. At a minimum, one Docker host is needed for the emulated IoT devices, and another for the virtual gateways to receive data from the IoT devices. On the application side, many Docker hosts can run different applications that support a streaming application. In our scenario, we have one `Apache Kafka`[5] broker to receive the data from the IoT gateways, one `Apache Spark`[6] instance for the streaming analytics, and one `Apache Cassandra`[7] instance to persist the data.

[3] https://github.com/brianr82/EMU-IOT
[4] https://www.docker.com/
[5] https://kafka.apache.org/
[6] https://spark.apache.org/streaming/
[7] http://cassandra.apache.org/

C. EMU-IoT Configuration and Customization

EMU-IoT was written in Python, therefore any reference to programming code will be in Python. The first step in configuring an IoT experiment is to provide details to the IoT Orchestrator.

```
IoTExperiment =
    IoTExperimentLinearRegression()
experiment_number ="1"
IoTExperiment.setExperimentName
("Linear_Regression_50")
IoTExperiment.setTargetCPUUtilization("50")
IoTExperiment.configureExperiment("mix")
IoTExperiment.
   set_max_dev_on_a_prod_host("200")
IoTExperiment.
   set_max_on_virtual_gateway("50")
IoTExperiment.run()
```

Figure 9: IoT Orchestrator Parameters

Figure 9 shows the configuration parameters of EMU-IoT for the experiment. This set of statements will execute a large scale experiment involving hundreds of temperature and camera devices, potentially across different geographic regions. In this example configuration, historical data will be used to compute a regression function to predict the number of IoT devices required to hit a target CPU utilization of 50% in our message broker.

D. Creating a custom experiment class

The primary function of the EMU-IoT lab is to simulate scenarios and to carry out actions that realize these scenarios. The `IoT Experiment` class is designed to drive a simulation by generating test cases that cause changes in the simulation of IoT networks. This class requires minor configuration depending on the needs of the user. For example, to make use of new libraries or to add more application hosts based on a proposed IoT application, all that is needed is to import those libraries and provide the IP addresses of the physical hosts. Details regarding how to modify the utility methods *configureNetwork(), configureMonitor(), IoTNodeSetup(), cleanup()* can be found on the GitHub repository.

The two important methods in the experiment class are *generateTestCase()* and *configureExperiment()*.

The *generateTestCase()* method drives the creation of the IoT devices. Test cases are changes in the scenario where we want to see if they have an effect on the resource metrics of the IoT application. In each test case, we create a specific quantity for each type of IoT device. As shown in Figure 10 (lines 2 and 9), two loops are used to create a specific number of temperature and light sensors, and IoT cameras in each iteration. The purpose of this approach is to keep generating test cases in a systematic manner to progressively increase or decrease the workloads so that the changes in the behaviour of the application can be observed. In the future, we intend to make the specification of test cases configurable through a UI instead of having to write these loops manually.

```
1 ’’’Generate Temp and Light Sensors’’’
2 for i in range
3 (’0’,’self.temperature_sensors_per_test_case’):
4    self.DeviceServiceTemperature
5    .addVirutalIoTDevice
        (’self.IoTLinearRegressionLoadbalancer’
6    ,’self.IoTLinearRegressionMonitorManager’)
7     self.target_active_producers =
         ’self.target_active_producers + 1’
8 ’’’Generate IoT Cameras’’’
9 for i in range
10 (’0’, ’self.camera_sensors_per_test_case’):
11   self.DeviceServiceCamera.addVirutalIoTDevice
        (’self.IoTLinearRegressionLoadbalancer’,
        ’self.IoTLinearRegressionMonitorManager’)
12   self.target_active_producers =
        ’self.target_active_producers + 1’
```

Figure 10: generateTestCase()

```
1 if experiment_type == ’mix’:
2    self.temp_sensors_per_test_case =
        int(round(10/11 *
        Regression(’training_data/mix’)
        .getGenerateTestCase
        (’self.targetCPUUtilization’)))
3    self.cam_sensors_per_test_case =
        int(round(1/11 *
        Regression(’training_data/mix’)
        .getGenerateTestCase
        (’self.targetCPUUtilization’)))
4    self.setApplicationToMonitor
5    (’IoTMonitorType.kafka’)
```

Figure 11: configureExperiment()

A new *if* statement in the *configureExperiment()* method can be added if a new type of experiment is required. As shown in Figure 11 (line 1), the name of experiment can be set. In this example, it is called *mix*, meaning its a combination of two different types of devices. This setting will result in an experiment that involves creating different types of virtual IoT devices. In Figure 11 (lines 2 and 3), we use the regression library and the training data to get the number of devices we should create in a test case. The next step in configuring the experiment is to determine what application to monitor when executing test cases. As shown in Figure 11 (lines 4 and 5), we are monitoring the resource metrics of the Kafka broker.

E. Creating new IoT Devices

Virtual IoT devices in EMU-IoT are encapsulated as Docker images. They are independent and self-contained just like physical IoT devices. Therefore, to create a new device for a specific application/experiment the proposed device needs to adhere to the minimum requirements as described in Section III-B. To recap, the requirements are to have a configuration interpreter, a data generator and a data forwarder. A program needs to be running inside the Docker container that can accept configuration arguments, application logic that will generate data similar to a physical IoT device, and a way to transmit that data. The configuration parameters are passed to the Dockerized IoT device in EMU-IoT as shown using Python on lines 4-9 in Figure 12.

```
1 self.IoTProducerBinding.NodeDockerRemoteClient
2 .containers.run("sensorsim:latest"
3 detach=True,
4 environment={'PI_IP':
      self.destination_gateway_ip,
5             'PI_PORT':
                  self.dest_virtual_gateway_port,
6             'NUM_MSG':
                  self.number_of_msg_to_send,
7             'SENSOR_ID': self.IoTDeviceName,
8             'DELAY':
                  self.producer_device_delay},
9             name=self.IoTDeviceName
10 )
```

Figure 12: Create an IoT Device

In our example, we created a temperature sensor, for which we wrote a program in C that emits integer values every second and does an HTTP POST to a virtual gateway. On the gateway side, we used Node-RED[8] to create a small footprint web server that receives this data and then forwards it to a message broker. More details can be found in our GitHub repository.

V. Results

In this section we present the results of the experiments that we ran based on the example IoT application described in Section IV. The purpose of these experiments is to demonstrate the experimental capabilities of EMU-IoT and how we overcome several of the challenges described in Section II. We tackle these challenges by deploying configurable devices across a geographically dispersed infrastructure that can be scaled to execute experiments with hundreds of IoT devices.

In these experiments we perform bottleneck detection at the Kafka message broker shown in Figure 13. We also present a brief cost analysis that shows how much time and money can be saved by using the emulator.

The experiments consist of two phases. First, historical data is collected to profile the application by performing an exhaustive search, in the form of load tests, measuring the CPU utilization of the running IoT application and the number of sensors that are driving traffic to that application. In the second phase, the historical data is used to build a performance model of the application and make predictions to determine the number of IoT devices that would cause the application to reach a particular target CPU utilization. Then, experiments are executed to validate the prediction accuracy by instantiating these devices and measuring the CPU utilization. While we examined only CPU utilization as an input to the model, it is certainly possible to include other factors such as disk IO, memory usage, etc.

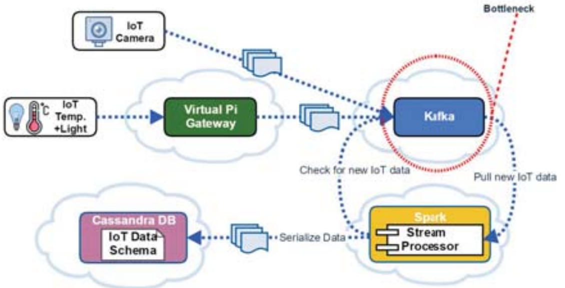


Figure 13: Example IoT Application

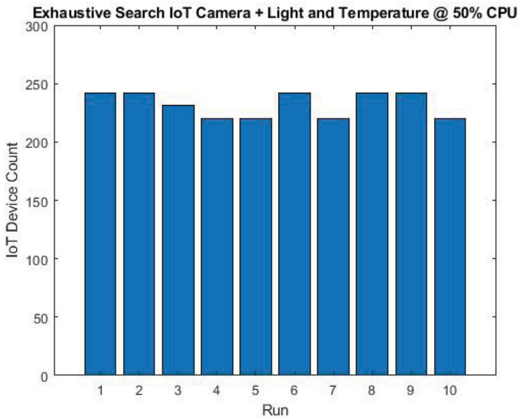


Figure 14: Exhaustive Search IoT Camera and Temperature + Light Devices

As shown in Figure 13 the experiment is heterogeneous, meaning we have two different types of IoT devices which have different traffic patterns. The Smart Testing Framework is used to generate test cases autonomically, with each test case consisting of 10 IoT temperature devices and 1 IoT camera. The results of the exhaustive search are shown in Figure 14. We executed 10 runs to search for the number of IoT devices required to consume 50% of the CPU resources at the Kafka broker. We can see that it requires approximately a total of 230 devices (210 IoT temperature + 20 IoT cameras) to consume 50% of the available CPU resources.

Using the data collected in the exhaustive search experiment, we computed a regression function as shown in Figure 15. We used this function to make predictions about the number of devices required to consume 60% and 70% of the CPU resources of the machine that Kafka is located on. The model predicted 301 and 362 devices, respectively.

We ran experiments with the predicted number of devices and measured the CPU utilization. The results of the experiments as depicted in Figure 16a and Figure 16b show that the model produces fairly accurate predictions as compared to the actual CPU utilization. The red line across the graph is the target CPU utilization target and the blue bars are the actual observed values on each of the 5 runs.

[8] https://nodered.org/

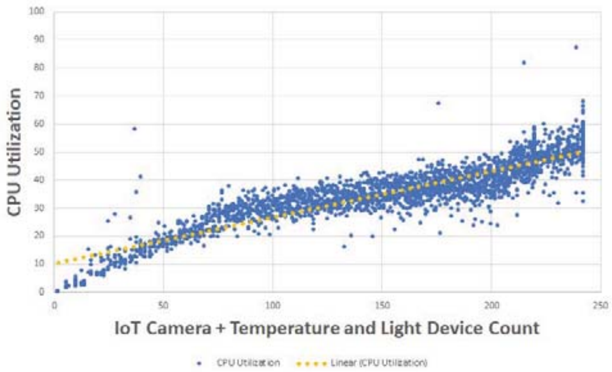
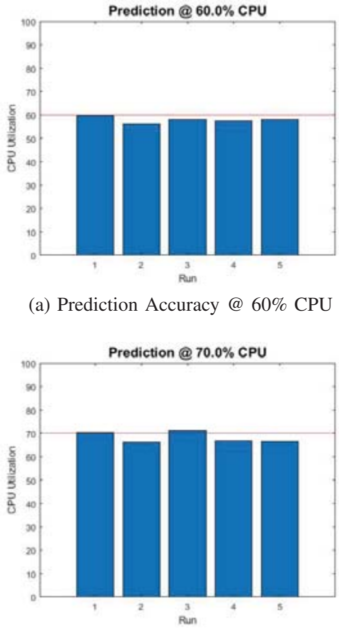


Figure 15: Regression IoT Camera and Temperature + Light Devices



(a) Prediction Accuracy @ 60% CPU

(b) Prediction Accuracy @ 70% CPU

Figure 16: Prediction Results

A. Cost Analysis

All the devices that were created in this experiment are virtualized which provides considerable cost savings over having to acquire the physical devices. For example, to run the 70% CPU usage prediction experiment a total of 362 IoT devices would have been needed. The total cost of running this experiment with typical consumer grade physical IoT devices would be $33,741.38 as shown in Table I[9] [10]. Devices that are industrial grade quality would be even more expensive. Also, there would be the additional costs of deploying the physical devices. As EMU-IoT can be quickly deployed in the cloud, the compute resources required to execute this experiment in Amazon EC2 were very minimal as shown in Table II.[11]

IoT Device Type	Unit Cost	Quantity	**Total Savings**
Temp and Light	$79.99	329	**$26,316.71**
Camera	$224.99	33	**$7,424.67**

Table I: IoT Device Savings

Instance Type	Quantity	$/hour	Hours	**Total Cost**
a1.2xlarge	2	$0.0394	8	**$0.63**
a1.xlarge	5	$0.0197	8	**$0.78**
a1.large	1	$0.0098	8	**$0.08**

Table II: Compute Resource Cost

VI. Related Work

There have been several initiatives towards creating tools that can reliably and accurately simulate IoT networks. Chernyshev identified 3 types of IoT simulators that are actively being researched [11]: Full stack simulators, Big Data Processing simulators and Network simulators. Full stack simulators are defined as simulators that provide end-to-end support for devices and applications. Big Data simulators focus on using cloud computing resources for big data processing in an IoT context. Lastly, Network simulators focus on network traffic and evaluating different protocols that are typically used in IoT networks. Our approach is to combine all three types of simulators to create a comprehensive solution.

In the full stack category, a simulator called DPWSim allows users to define and create simulated IoT networks [2]. It provides a robust set of tools, however the platform is limited to using DPWS (Devices Profile for Web Services) standards which is based on WSDL. Our goal is to create a tool where any communication standard can be used. In addition to singular protocol being used in DPWSim, it also implements the SOAP protocol, which is mainly used in defining message exchange rules between enterprise applications. SOAP messages incur a large amount of overhead by design. IoT protocols are typically designed to be lightweight due to limited available processing power and bandwidth. In our work, the goal is to implement emulated devices that use IoT type protocols such as messaging over MQTT or HTTP. Also, in the full stack category we have iFogSim which allows for end-to-end simulation of devices, edges and the processing infrastructure for IoT networks [12]. While this toolkit provides all the features of a simulation environment, it does not use the actual hardware to execute the simulation making the results of experiments difficult to compare with a physical system. Our work improves upon this model by providing a tool that can be executed on the actual network where the real system will run on.

In the big data simulator category, there is IoTSim and SimIoT. In the case of IoTSim, it does provide the capability

[9] https://www.lorextechnology.com/4k-security-camera/4k-ultra-hd-8mp-noctunnal-ip-camera/LNB8921BW-1-p

[10] https://store.ubibot.io/collections/the-ubibot-product-family/products/ubibot-ws1

[11] https://aws.amazon.com/ec2/spot/pricing/

to simulate large scale IoT networks but has two major drawbacks [4]. The first limitation of IoTSim is that the workload generation process is only based on the MapReduce model. While this is a common scenario when processing IoT data, many other types of scenarios exist. For example, image recognition of live video streams requires a completely different set of steps necessary to analyze the data as compared to batch processed numerical data such as those generated from environmental sensors. Moreover, this limitation leads to the inability to emulate heterogeneous IoT networks, since they can only support devices that generate MapReduce type data. In our approach, we have an emulator that is capable of evaluating the performance of *any* IoT big data application as long as it can be containerized. The second limitation of IoTSim is that it cannot be executed in the actual environment as it is also an extension of CloudSim, meaning it has the same limitation as the previously mentioned simulator, iFogSim. This is a significant drawback, because big data applications all behave differently depending on what type of hardware is available to them. In the case of SimIoT, this platform also lacks support for heterogeneous IoT networks [13]. This is a key characteristic of IoT networks, where many types of devices are present. As previously mentioned, in our approach, the platform currently supports any type of software-defined version of an IoT device without requiring changes to the underlying platform. SimIoT also lacks the ability to track QoS metrics to drive network optimization such as adding or removing computing resources. In our approach, we solve this by implementing a smart testing framework.

Network simulators are also being repurposed for IoT simulation.. Network simulation is a well-researched area that predates IoT research by several decades, so it makes sense that tools originally designed to simulate network traffic are being extended to support IoT network research. One such popular tool is CupCarbon, which has been extended to include IoT features for emulation [1]. While this emulator can execute a simulation on real hardware, it is limited to running on Raspberry Pi and requires modification to work on any new platform. One of our goals is to make our emulator platform agnostic, so that it can be installed on any cloud provider. Another popular tool is Cooja which is used primarily for simulating wireless sensor networks [3]. This simulator overcomes the limitations of CupCarbon but it only allows simulation of the devices and not the applications that process the sensor data. This necessitates the need for another simulator to handle the data ingestion functions, thus making it difficult to evaluate an end-to-end scenario on an IoT network.

VII. Conclusion and Future Work

We presented the design and implementation of a new platform that provides users with an environment to execute and experiment with large scale IoT networks without the high capital costs associated with a physical deployment. We provided a methodology for designing reliable and configurable software defined virtual sensors. We also provided a systematic process for using the virtualized IoT devices along with IoT applications to stress test IoT networks from end to end.

We intend to support new IoT device types by using the generic framework we defined in EMU-IoT for creating new devices. An example of such devices could be vehicles that generate a variety of data from a collection of sensors. Our architecture is currently capable of processing this type of data. Other device types we plan to support are IoT devices that are configurable at runtime. This would give us the ability to change the behavior of the device according to an operation plan. An example of this type of device would be a smart home thermostat. Moving forward into the future, the number and variety of IoT devices is expected only to grow. To accommodate this growth, we have designed our platform to be extendable to model these future scenarios.

References

[1] A. Bounceur, O. Marc, M. Lounis, J. Soler, L. Clavier, P. Combeau, R. Vauzelle, L. Lagadec, R. Euler, M. Bezoui, and P. Manzoni, “Cupcarbon-lab: An iot emulator,” in *2018 15th IEEE Annual Consumer Communications Networking Conference (CCNC)*, Jan 2018, pp. 1–2.

[2] S. N. Han, G. M. Lee, N. Crespi, K. Heo, N. V. Luong, M. Brut, and P. Gatellier, “Dpwsim: A simulation toolkit for iot applications using devices profile for web services,” in *2014 IEEE World Forum on Internet of Things (WF-IoT)*, March 2014, pp. 544–547.

[3] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, “Cross-level sensor network simulation with cooja,” in *Proceedings. 2006 31st IEEE Conference on Local Computer Networks*, Nov 2006, pp. 641–648.

[4] X. Zeng, S. K. Garg, P. Strazdins, P. P. Jayaraman, D. Georgakopoulos, and R. Ranjan, “Iotsim,” *J. Syst. Archit.*, vol. 72, no. C, pp. 93–107, Jan. 2017. [Online]. Available: https://doi.org/10.1016/j.sysarc.2016.06.008

[5] G. Kecskemeti, G. Casale, D. N. Jha, J. Lyon, and R. Ranjan, “Modelling and simulation challenges in internet of things,” *IEEE Cloud Computing*, vol. 4, no. 1, pp. 62–69, Jan 2017.

[6] B. Ramprasad, J. McArthur, M. Fokaefs, C. Barna, M. Damm, and M. Litoiu, “Leveraging existing sensor networks as iot devices for smart buildings,” in *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*, Feb 2018, pp. 452–457.

[7] M. Chan, E. Campo, D. Estève, and J.-Y. Fourniols, “Smart homes — current features and future perspectives,” *Maturitas*, vol. 64, no. 2, pp. 90 – 97, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0378512209002606

[8] M. Vögler, J. M. Schleicher, C. Inzinger, and S. Dustdar, “Diane - dynamic iot application deployment,” in *2015 IEEE International Conference on Mobile Services*, June 2015, pp. 298–305.

[9] G. Rawal, “Costs, savings, and roi for smart building implementation,” Jun 2018. [Online]. Available: https://blogs.intel.com/iot/2016/06/20/costs-savings-roi-smart-building-implementation/#gs.GGIJbNcg

[10] B. Ramprasad, J. Mukherjee, and M. Litoiu, “A smart testing framework for iot applications,” in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, Dec 2018, pp. 252–257.

[11] M. Chernyshev, Z. Baig, O. Bello, and S. Zeadally, “Internet of things (iot): Research, simulators, and testbeds,” *IEEE Internet of Things Journal*, vol. 5, no. 3, pp. 1637–1647, June 2018.

[12] H. Gupta, A. VahidDastjerdi, 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. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2509

[13] S. Sotiriadis, N. Bessis, E. Asimakopoulou, and N. Mustafee, “Towards simulating the internet of things,” in *2014 28th International Conference on Advanced Information Networking and Applications Workshops*, May 2014, pp. 444–448.