CCBD - IoTSim-Edge

Sriram S K
PES1201700097
PES University
Bengaluru

Kevin Arulraj
PES1201700659
PES University
Bengaluru

Shreyas Prashanth
PES1201701249
PES University
Bengaluru

Ashwath Janardhanan
PES1201701121
PES University
Bengaluru

***Abstract*—IoTSim-Edge is a simulator that allows one to simulate IoT devices and end devices of their own preference in an environment governed by the user. IoTSim-Edge is built on the top of the CloudSim simulation library which provides the underlying mechanisms for handling communication. The IotSim library provides various resources in order to realistically simulate an IoT environment. Some of these tools include different types of IoT devices (e.g. car sensor, motion sensor), movable sensors and so on. In this project, we have understood and extended some of the functionality of this tool.**

***Index Terms*—edge computing, cloud, simulator, cloudsim**

I. Introduction

CloudSim [1] is a simulation tool that allows cloud developers to test the performance of their provisioning policies in a repeatable and controllable environment, allowing for easy experimentation and design of algorithms in what would otherwise be a costly process. It is a library for the simulation of cloud scenarios. It provides essential classes for describing data centres, computational resources, virtual machines, applications, users, and policies for the management of various parts of the system such as scheduling and provisioning. The CloudSim Core simulation engine provides support for modeling and simulation of virtualized Cloud-based data center environments including queuing and processing of events, creation of cloud system entities (like data center, host, virtual machines, brokers, services, etc.) communication between components and management of the simulation clock.

Advances in IoT have a transformative impact on society and the environment through a multitude of application areas including smart homes, smart agriculture, manufacturing and healthcare. To achieve this, an ever increasing number of heterogeneous IoT devices are continuously being networked to support real-time monitoring and actuation across different domains. Traditionally, the enormous amount of data, generally known as the big data, is sent to the cloud by IoT devices for further processing and analysis.

However, the centralised processing in cloud is not suitable for numerous IoT applications due to the following reasons:

1) Some applications require close coupling between request and response.
2) Delay incurred by the centralised cloud-based deployment is unacceptable for many latency-sensitive applications.
3) There is a higher chance of network failure and data loss.
4) Sending all the data to cloud may drain the battery of the IoT device at a faster rate.

The introduction of edge computing addresses these issues by providing the computational capacity in a near proximity to the data generating devices. Smart edge devices such as smartphones, Raspberry Pi etc. support local processing and storage of data on a widespread but smaller scale. However, the constituent devices in edge computing are heterogeneous as each one may have specific architecture and follows particular protocols for communication.

Evaluating the framework in a real environment gives the best performance behaviour, however, it is not always possible as most of the test frameworks are in development. Even if the infrastructure is available, it is very complex to perform the experiment as setting it up requires knowledge of all the associated IoT and edge devices which is not intuitive. Additionally, performing the experiment in the real environment is very expensive due to the set up and maintenance cost incurred. To overcome these issues, a feasible alternative is the use of simulators . The simulators offer a window of opportunity for evaluating the proposed hypothesis (frameworks and policies) in a simple, controlled and repeatable environment. A simulation environment must mimic the key complexity and heterogeneity of real networks and support multiple scenarios that affect real IoT deployments.

II. Installation and Documentation

IoTSim-Edge [2] is one of the few simulators which solves the many complexities of simulating an edge computing environment. Other simulators like EdgeCloudSim [3] and iFogSim [4] address many but not all of the intrinsic challenges. These challenges are mentioned below and explored in detail in the attached document.

IotSim depends on Cloudsim and a few other libraries like javafx, lombok etc. getting the dependencies in order and understanding the maven build system to get a fully functional IotSim installation was a nontrivial task. In addition, since IotSim is built on top of cloudsim and the main simulation is handled by cloudsim, it is essential to have the source of cloudsim for understanding. Thus the provided cloudsim.jar did not prove helpful and we built cloudsim from source so that we could create our own jars for custom functionality. A brief description of the installation process is mentioned:

IoTSim-Edge requires a few basic prerequisites before installing onto your local system. And they are:

- Java JDK - if your JDK version is ¿= 11, the GUI will not be available directly as the JavaFX runtime library has been removed from the main JDK. Thus, for later versions of the JDK, JavaFX must be separately installed and included in the project.
- Apache Maven
- A suitable Java IDE for easy development (e.g. IntelliJ IDEA)

One must clone DN Jha's IoTSim-Edge [5] Github repository and open the project with an IDE. The cloudsim.jar library needs to be added to the CLASSPATH and the Lombok plugin too is to be installed.

Upon rebuilding the project and running the test example programs we can verify if IoTSim Edge runs seamlessly. Finally, run the `Main.java` file and if a dialog box GUI appears it means that IoTSim Edge has been successfully installed.

If a custom Cloudsim is required, it must be compiled from source and replaced with the default cloudsim in IoTSim-Edge.

Since this project will be carried forward in the years to come, it is of utmost importance to have a clear guide to installing IoTSim, especially since IoTSim's documentation is lacking in that regard. Thus, we have written a step by step guide to installing IoTSim using the Intellij iDEA IDE.

In order to kickstart development and enhance our own understanding, we have compiled documentation on iotsim's architecture and how it maps and extends on cloudsim's architecture. It is briefly described here and a more detailed version is attached.

III. Architecture of IoTSim-Edge

IoTSim-Edge is built on the top of CloudSim simulation tool. CloudSim provides the underlying mechanisms for handling communication among subscribed components (e.g. broker, edge datacenter, IoT resources) using an event management system. The core components of CloudSim are extended to represent the edge infrastructure in line with edge's features and characteristics.

Traditionally, cloud resources are used for processing IoT data. But in Edge-IoT approach, sensor data is processed in the edge datacenter for faster processing time. Edge DataCenter consists of heterogeneous processing devices such as smartphones, laptop, Raspberry Pi and single server machine. Users can provide inputs through a graphical user interface (GUI) by mentioning different device configuration and policies. Edge-IoT management layer consists of several components such as Edgelet, policies, mobility, battery, synchronism, QoS, network protocols, communication protocols, transport protocols, and security protocols.

For the implementation of IoTSim-Edge, existing classes of CloudSim are extended as shown below as well as numerous new classes are defined in order to model realistic IoT and edge environments. Any entity that extends SimEntity class can seamlessly send and receive events to other entities when required through the event management engine.

For modelling an edge infrastructure, new classes are designed and implemented as shown and explained below. The main classes are EdgeDataCenter, EdgeBroker, EdgeDatacenterCharacteristics, EdgeDevice, MicroELement, and EdgeLet.

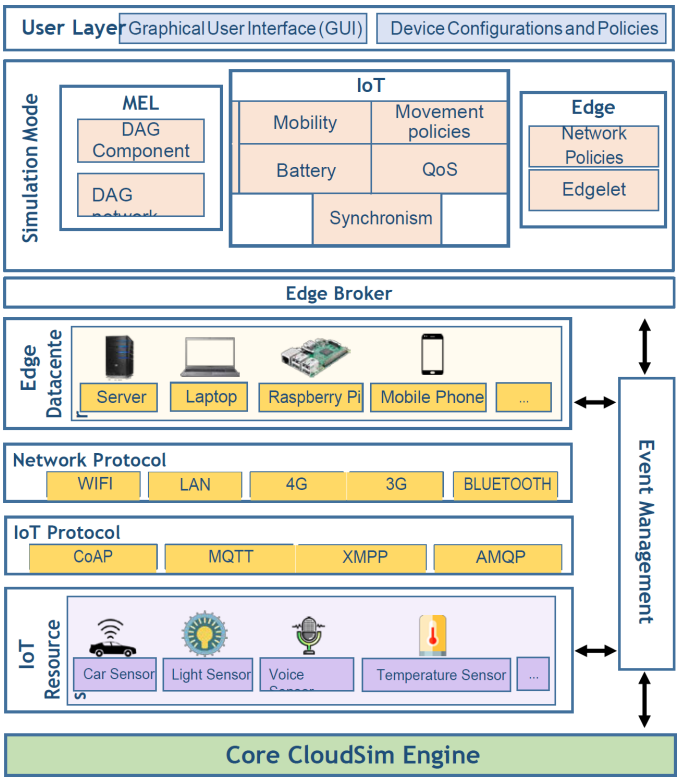

Fig. 1. The architecture of IotSim-Edge

- *EdgeLet* - This class models the IoT generated data and MEL processing data. Once an IoT device establishes connection with its respective edge device(s), it generates IoT sensed data as required in a form of EdgeLet.
- *EdgeDataCentre* - It intercepts all incoming events and performs different operations based on the payload of the event, such as resource provisioning and submitting EdgeLet requests to respective MEL(s).
- *EdgeBroker* - This class is a users' proxy, in which it generates users' requests in accordance with their prescribed requirements. Its duties include requesting IOT devices to send data to their respective edge devices and receiving processing results from MEL.
- *EdgeDevice* - It hosts several MELs and facilitates the procedure of CPU sharing mechanism via a given CPU sharing policy.
- *IoTDevice* - This class models the core characteristics of IoT devices. Any new required type of IoT device can easily extend the IoTDevice Class and implement the new features. This class uses IOTProtocol and NetworkProtocol classes to categorize any IOT device.

- *MEL* - This class represents one component of the IoT application graph. It represents the main processing requirement of the application component.

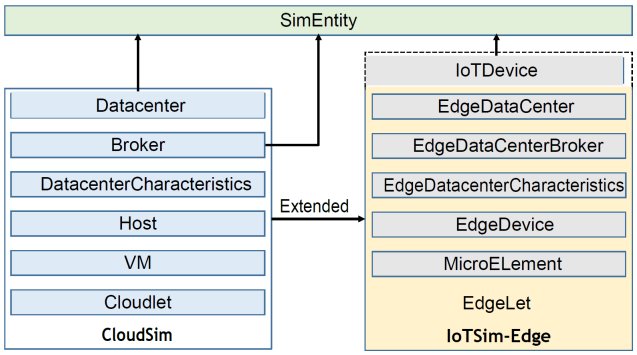

Fig. 2. Mapping of Cloudsim and IotSim classes

IV. Challenges Solved by IoTSim-Edge

IoTSim-Edge tackles the many challenges of an IoT environment in novel ways. It uses a graph modeling abstraction with MicroELements (MELs), allowing us to model various kinds of applications. It has a high level of abstraction allowing us to combine heterogeneous IoT devices and all their configuration parameters (battery capacity, mobility etc.) for real-world applications.

In essence, it provides a generic model that can be adapted for the specifics of any application using a number of IoT devices which in turn use a number of configurations varying in protocol, battery capacity, mobility etc.

Cloud computing is a model for on-demand access to a shared pool of configurable resources (e.g. compute, networks, servers, storage, applications, services, and software). Cloud computing platforms are well suited for hosting IoT applications as they offer an elastic hardware resources (e.g. CPU, Storage, and Network) that can be scaled on-demand for handling large quantities of data from IoT applications with uncertain volume, variety, velocity, and query types.

These two technologies are inherently and increasingly getting entwined with each other. Because of this, evaluation and analysis of IoT applications in a real cloud computing environment can be a challenge for several reasons:

- It is not cost-effective to procure or rent a large scale datacenter resource pool that will accurately reflect realistic application deployment and let practitioners experiment with dynamic hardware resource and big data processing framework configurations, and changing data volume, velocity, and variety.
- Frequently changing experiment configurations in a large-scale real test bed involves a lot of manual configuration, making the performance analysis itself time-consuming. As a result, the reproduction of results becomes extremely difficult.
- The real experiments on such large-scale distributed platforms are sometimes impossible due to multiple test runs in different conditions.
- It is almost impractical to set up a very large cluster consisting of hundreds or thousands of nodes to test the scalability of the system.

An obvious solution to the aforementioned problems is to use a simulator supporting IoT application processing. A simulator not only allows us to measure scalability of computing resources for IoT applications efficiently, but also enables us to determine the effects of various variables like datacenter configuration, the numerous types of IoT devices and their communication protocols etc.

However, there are quite a few challenges in developing an IoT simulator as well due to the heterogeneous nature of the environment. Some challenges and the manner in which IoTSim-Edge solves them are:

- ***Modeling networking graph between diverse type of IoT and edge computing device in an abstract manner can be very challenging***
  NetworkProtocol class presents the modeling of network protocols (e.g. WiFi, 4G LTE). Implementing such models in the IoTSim-Edge framework is required to properly evaluate the performance of IoT-Edge applications. Each network type is designated with its relative network speed (e.g. 200 Mbps for WiFi, 150 Mbps for 4G LTE). By modeling the transmission rate, transmission time can be taken into account the EdgeLet size. Due to the distributed nature of IoT, there are a large number of devices present, connected in necessarily complicated ways. A suitable manner of representation is required to model any non-trivial application. IoTSim solves this issue by abstracting the connections and making them configurable through the use of a JSON file, which specifies how the devices are connected in a simple and human-readable manner.
- ***Capacity planning across edge computing layers is challenging as it depends on various configuration parameters including data volume, data velocity, upstream/downstream network etc.***
  To solve this issue, IoTSim utilizes the features of one of the most popular cloud simulators, Cloudsim. Similar to how Cloudsim allows for configuration of parameters such as network latency, data volume and velocity. Cloudsim implements two schedulers for resource allocation: VM scheduler allocates VMs to host while the Cloudlet scheduler allocates different tasks to VMs. IoTSim's EdgeDataCenter class is responsible for establishing connection between edge and IoT devices based on the given IoT protocol (e.g. CoAP) along with performing edge resource provisioning, scheduling policies, and monitoring edge processing. IoTSim builds on the same to provide flexible capacity planning across the edge computing layer.
- ***Modeling data and control flow dependencies between IoT and edge layers to support diverse data analysis work-flow structure is non-trivial.***
  Two types of IoT nodes could be modelled and config-

ured, i.e. sensors and actuator nodes. Sensing nodes will collect the information of surroundings through sensors and send the information for processing and storage. Actuators will be activated based on the analysis of the data. The communication layer is responsible for data transfer to/from IoT devices, edge devices and cloud. Employing combinations of these various node types enables the modelling of all topologies and architectures of IoT systems. IoT nodes are configured with a power source that could be a battery, USB charging point or continuous power supply. Battery consumption is tracked during the simulation for real deployments. Nodes are associated with different types of connections (e.g. 3G, Bluetooth, WiFi) and tracked signal strength.
Possible data of each sensor are stored in a JSON file, and the sensors are configured to read data from their files and submit a reading at each time interval. Data could be selected sequentially, randomly, or randomly within a specific range according to the hypothetical scenarios.

V. Adding missing functionality to IoTSim

To become better acquainted with the simulator, we added additional functionality to the TemperatureSensor. Originally, the packets generated by the IoT devices (Edgelets) do not actually contain data. They are of a specified size (which signifies how long they need to be processed) but there is no data in the packet. To simulate a basic application and become better acquainted with modifying IotSim, we extended the TemperatureSensor class to generate a random junk value as a part of every Edgelet. These values are parsed by the EdgeDevices when they are processing the edgelet and on receiving extreme values, the data is logged. The idea being that in a real application, extreme values would be caught and acted upon.

In many places, the behaviour of the simulator was not expected. Configuration of various parameters through the json file had no effect on the simulation whatsoever. On closer inspection, we noticed that in some cases, the values were hardcoded or even ignored. These include :

- SimpleMovingPolicy - This class handles the mobility of the IoT devices, an important part of the simulator. The configuration file of the simulation allows for movement across a 3-dimensional space, however, the SimpleMovingPolicy only took the values for `x` for the simulation, ignoring movement across the other two dimensions. Fixing this involved refactoring code in this class as well as in the simulation driver code.
- DataShrinkFactor - As mentioned earlier, IotSim simulates an application through an abstraction called MicroElements (MELs). These MELs are characterized by a variable known as the datasizeShrinkFactor which controls the degree of processing happening on the edge device. Higher this factor, the higher the amount of data that is transferred instead of being processed on the edge device itself. Previous research has shown that battery is drained faster by transmission than by computation and thus it is preferable to have a low shrinking factor to extend battery life. Conversely, if latency is the constraint, a high datasizeShrinkFactor is preferred as less data needs to be transferred.

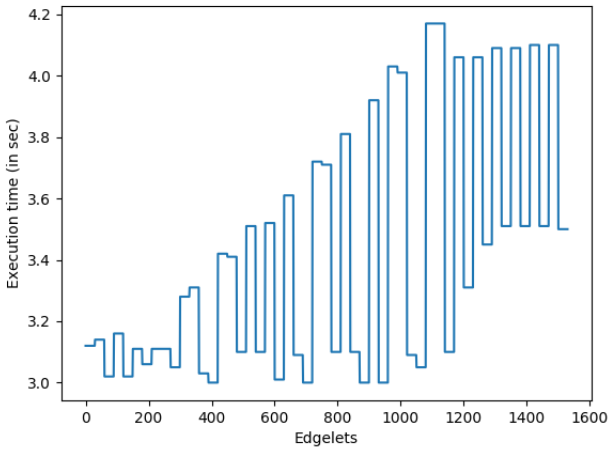

Fig. 3. The execution time for the edgelets

VI. Case Study - Healthcare System

We use the existing IoTSim framework for a case study of a healthcare system. For this simulation, multiple parameters are varied to find the best solution for this hypothetical scenario.

A system of many moving IoT devices (to be considered the equivalent of 'FitBits'), which can process data. These devices move in a specified direction while transmitting data. The simulation parameters are configured in multiple ways and the effect of each parameter on the simulation results (queue time, battery duration etc.) is observed.

The system is constructed in the following manner : 30 IoT devices in groups of ten. Each group moves from (0,0,0) along a particular axis over the course of the simulation. The Edge Device being used is a Raspberry Pi.

The results for the base simulation are shown in **Fig. 3**:

The mean execution time is 3.422 seconds with the variance being 0.160. Considering that this is not a time critical application, a latency of around 3.5 seconds is acceptable.

Furthermore, since the edge devices are also battery powered, we have tested how the battery life of the edge devices varies with different shrink factors. The shrink factor is a measure of the degree of computation to transmission. If the shrink factor is low, then a larger portion of the data is processed on the edge device.

Finally, the choice of network/communication protocol is considered. The network protocol chosen is Bluetooth, specifically a low energy version of Bluetooth called BLE (Bluetooth Low Energy). BLE's energy efficiency and low data rate has made it one of the more popular protocols in many applications.

There are four communication protocols available in IoTSim. They are as follows:

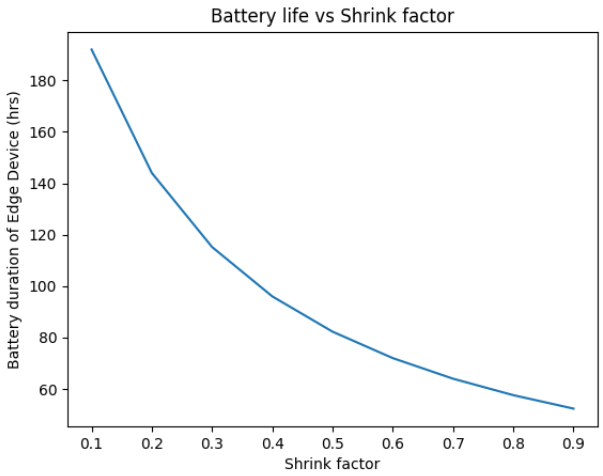

Fig. 4. The battery life of the Edge Device with different Shrink Factors

Protocol Name	Transmission Speed	Battery Drain Rate
MQTT	1	1
AMQP	1	1
COAP	3	1
XMPP	3	1.5

We can see that COAP appears to be a little better than XMPP which is the default. However, XMPP has a higher drainage rate due to the fact that it is built over TCP, offering it fault tolerance as compensation. In this case however, since messages are continuously transmitted, it is okay to use UDP as base protocol and thus we switch to COAP, allowing the battery of the IoT devices to last longer.

VII. Extending IoTSim-Edge

A. A new VmAllocationPolicy

The VmAllocationPolicy being used by default was the *VmAllocationPolicySimple* which is a basic allocation policy. It takes into account all the hosts in a datacenter, and goes on allocating VMs on the basis of equal processing element usage. Each host has a number of processing elements (i.e. mips) available and this policy ensures that the VMs are allocated to the host with the most free processing elements.

In an edge computing environment, an additional important factor which arises is the distance. The distance from the EdgeDevice to the IoT sensor should be minimized, to control the latency of the application. Thus we devised a new VmAllocationPolicy class, the *VmAllocationPolicyEdge*.

While the simple VmAllocationPolicy does not take any parameters, here we take in two parameters, to be passed in through the JSON file. These are the priorities for the distance and the number of processing elements, **distPrio** and **pePrio** respectively.

VmAllocationPolicySimple sought to choose the host with the most free processing elements. Meanwhile, VmAllocationPolicyEdge maximizes the following function:

$$f(x) = pePrio * pe(x) + distPrio * dist(x)$$

Where, x belongs to the list of hosts, $pe(x)$ represents the amount of free processing elements of that host and $dist(x)$ represents the inverse of the distance of the host from the *average location of all the IoT devices*. The inverse is chosen because maximizing the inverse of the distance is the same as minimizing the distance itself.

Thus, we have a complete VmAllocationPolicy which allocates VMs based on a linear combination of the inverse distance from the centroid of the IoT devices and the number of free processing elements available at that host, with user defined priorities assigned to both components.

We experimentally verified the working of the new policy with the following setup for both the allocation policies. The setup of the simulation was as follows:

A single IoT device located at (100,100,100). Two hosts were created at the locations (0,0,0) and (100,100,100). Three VMs were to be allocated requiring the same number of processing elements. Both hosts were given an equal number of processing elements.

- Using VmAllocationPolicySimple - Using the default policy, all distances were ignored and allocations were made on the basis of processing elements alone. Since they were equal, the first Vm was allocated to Host1, the second to Host2, and the third to Host1 again.
- Using VmAllocationPolicyEdge - The priorities assigned to the two factors were 0.5, i.e. equal importance. Since Host2 is at the same location as the IoT device, the distance between them is zero. Or in other terms, the inverse distance is ∞. Thus, all 3 VMs were allocated to Host2.

B. A new CloudletScheduler

The CloudletScheduler controls the execution order of the cloudlets in a VM. The default cloudlet scheduler, *CloudletSchedulerTimeShared* does so in a simple manner. The MIPS available in the VM are split equally among all the cloudlets in the execution list and they are executed in parallel.

In an Edge Computing environment, however, this behaviour of treating each cloudlet equally is not always preferred. Cloudlets come from varied sources and some of them may require a lower latency while others are more elastic in nature. For example in a smart farm, considering moisture and light sensors, it would be useful to assign a higher priority to the moisture sensors, as soil moisture varying can be a cause for alarm. Similarly, different sensors may have different latency requirements and it would be useful to have a framework which allows us to discriminate between sensors as according to our needs.

This was the motivation behind the *CloudletScheduler-TimeSharedEdge* scheduler. When instantiated, it takes in a *sensorMap* i.e. a HashMap of priorities of the different sensors in the simulation.

During execution, the original CloudletSchedulerTimeShared calculates the *capacity* for each cloudlet in the following manner: the total available mips of the VM is divided by the sum of the number of CPUs required by each cloudlet, which in most cases is 1, thus simplifying down to number of mips by number of cloudlets.

This would ensure that each cloudlet to be executed gets a fair share of the mips available and the scheduler is completely ignorant of the type of the cloudlet i.e. from which sensor it was generated.

The new CloudletScheduler, on the other hand, works in a similar fashion. While calculating the the share of the mips for each cloudlet, the capacity is given by the total available mips divided by the *sum of the priorities* for each type of sensor. Similarly, while calculating the update for each cloudlet, this capacity is multiplied by the priority to give us a higher share for the cloudlets which are from high priority sensors, thus allowing us to process the high priority cloudlets quickly.

An example to illustrate the scheduler's working:

Assume two cloudlets are to be executed in a VM with mips available being 10000. For the default scheduler, the mips is split equally among both cloudlets, i.e. the capacity is 5000.

Now, let us assume that the two cloudlets are Temperature and Light cloudlets respectively, and that the defined priorities are 99 and 1. The sum of the priorities is 100. The capacity is now 10000/100 = 100. While this is the actual capacity, the effective capacity is given by:

$E(t) = A * sensorMap(t)$

where E is the effective capacity, A is the actual capacity, t is the type of the cloudlet and $sensorMap$ being the HashMap of sensor types and priorities.

Thus the effective capacities are:

E(light) = 100*1 = 100 E(temp) = 100*99 = 9900

Finally, we experimentally verified the working of the new Scheduler with a simulation whose setup is as follows:

20 IoT devices, 10 Temperature sensors and 10 Light sensors are instantiated. A single VM is used for simplicity and tested with both schedulers. The priorities were 1 and 200 for temperature and light respectively.

Scheduler	Avg LightSensor Execution Time	Avg TempSensor Execution Time
CloudletSchedulerTimeShared	0.623	1.270
CloudletSchedulerTimeSharedEdge	0.566	1.596

ACKNOWLEDGMENT

We would like to thank Prof. Phalachandra and the Center for Cloud Computing and Big Data for giving us this opportunity to work on the IotSim Edge and Clousim tools and improve our understanding of processing at scale in the cloud and edge environments.

REFERENCES

[1] Calheiros, Rodrigo N., et al. "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms." Software: Practice and experience 41.1 (2011): 23-50.

[2] Jha, Devki Nandan, et al. "IoTSim-Edge: A Simulation Framework for Modeling the Behaviour of IoT and Edge Computing Environments." arXiv preprint arXiv:1910.03026 (2019).

[3] Sonmez, Cagatay, Atay Ozgovde, and Cem Ersoy. "Edgecloudsim: An environment for performance evaluation of edge computing systems." Transactions on Emerging Telecommunications Technologies 29.11 (2018): e3493.

[4] Gupta, Harshit, et al. "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 47.9 (2017): 1275-1296.

[5] Jha, Devki Nandan, "IoTSim-Edge: A Simulation Framework for Modeling the Behaviour of IoT and Edge Computing Environments" https://github.com/DNJha/IoTSim-Edge