Politenico di Milano

Dipartimento Elettronica, Informazione e Bioingegneria

HEAPLab Project Report

EdgeCloudSim Report

Author:
Zhang Qiaolun

Supervisor:
Michele Zanella

March 9, 2019

Abstract

The simulation tool EdgeCloudSim provides environment to simulate edge computing scenarios, which can be used to conduct experiments that considers both computational and networking resources. However, the application allocated and scheduled in the system is atomic, which means the application cannot be divided and can only be allocated in one devices. But seperating the application into multiple kernels and distributing them to edge servers in the vicinity really counts because edge server is usually resource limited and may not capable to execute the application. In this project, the atomic application simulation in EdgeCloudSim is extended into thread-based application, which provides a more realistic simulation of computation and networking in edge computing.

1 Introduction

Edge Computing is a new computing paradigm which utilize the resources near the data source and collabrates with cloud computing to provide better services to the user. EdgeCloudSim is a simulation tool which can simulate edge computing and cloud computing scenarios which combine Edge Computing and Cloud Computing. EdgeCloudSim. EdgeCloudSim is extended from CloudSim and uses the discrete event management framework to simulate the dispatching and scheduling of applications.

The core simulation process is a loop function which checks the events related to all the entities and processes the event. Each entities can process different kinds of events. Figure 1 shows the relations between the EdgeCloudSim modules. The SimSettings class in core simulation modules will load and store all the simulation settings information, which includes the application parameters, the edge server settings, the simulation environment settings. The load generator module will utilize the information in SimSettings instance to generate applications that will be scheduled. Mobility module generates the position of each application when generating applications. Then the simulation of edge computing scenarios starts and the program will goes into a loop. At each cycle, the simulation will check whether there are some events starts at that time. MobileDeviceManager will check if there are new applications that need to be submitted at that time. The sttart time of each application is defined in load generator module before. If there is new applications that need to be submitted, edge orchestrator module will utilize

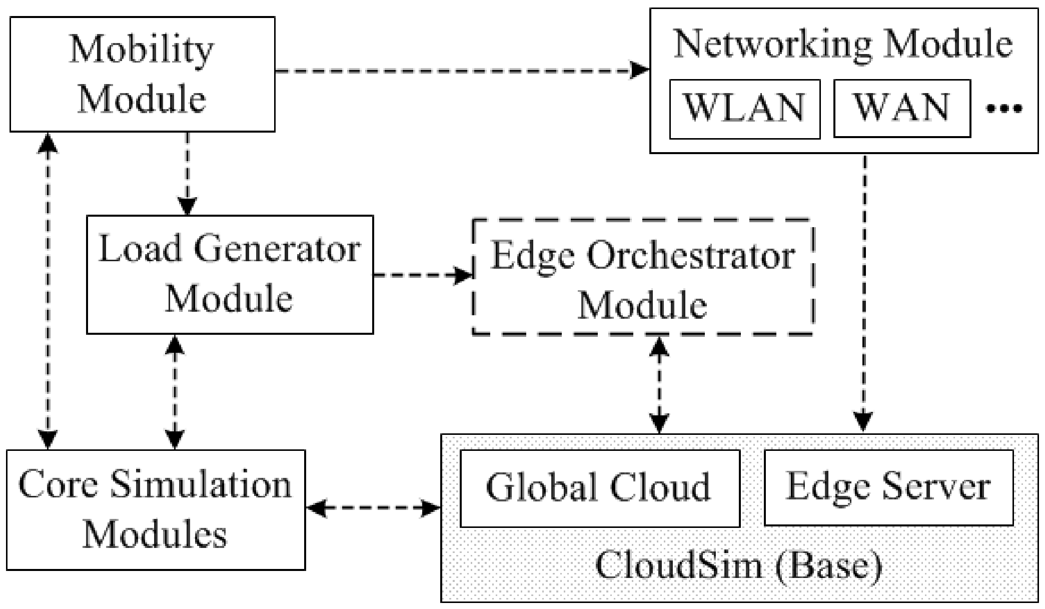


Figure 1: Module relationship between modules in EdgeCloudSim.

the specified allocation strategy to calculate the virtual machine in a server to execute the application. Then MobileDeviceManager class will utilize the calculated information in edge orchestrator module to submit the application to CloudSim (Base). The entity related to scheduling applications in the virtual machines is defined in CloudSim.

2 Problem Statement

In EdgeCloudSim, the application is atomic and can only be executed in one edge devices. However, some applications can be divided into multiple kernels and simulate the execution of these kernels really counts. Since single edge sever is resources-limited and may not have enough resoruces to execute the application. But executing the application in the cloud may lead to time-delay and even miss the deadline of the application. Consequently, application is divided into multiple small kernels, which are distributed into multiple edge server. So it is necessary to provide kernel-based application feature to simulate edge computing scenarios. The modules that need to be modified or added are listed as follows:

1. core: change the core classes related to load settings, managing events in the system and store status of kernel-based applications. This module also defines a KernelBasedAppliationStatus.java class to store the status of all the applications.

2. edge_clinet: define the classes for kernel.

3. mobility: change the mobility module to adapte to generate the positions for kernel-based applications.

4. task_generator: generate kerne_based applications

5. utils: define the classes related to kernel-based applications. Moreover, SimLogger.java are modified to calculate the simulation results for kernel-based applications.

6. config: the config file applications.xml are redifned for kernel-based applications.

3 Design and Implementation

3.1 Kernel-Based Application Design

In EdgeCloudSim, application is atomic and cannot be divided into smaller kernels. In this project, kenel-based application is introduced into this simulation tool. Kernel-based application is composed of a single kernel or multiple kernels. And the kernel is distinguised by kernel ID in this implementation. The original application.xml file are modified to represent the parameters for kernel-based application. The Applications.xml describes the types and parameters of applications in the simulation. Parameters in the original Applications.xml file of EdgeCloudSim is not explained, which makes it difficult to understand the meaning of these parameters. The meanings of the some important parameters are listed as follows:

1. usage_percentage: The usage percentage here is used to decide the percentage of this type of application in all of the applications generated. So add up usage_percentage field in the applications.xml, we can get 100 because we have sumed up all the percentages of all applications.

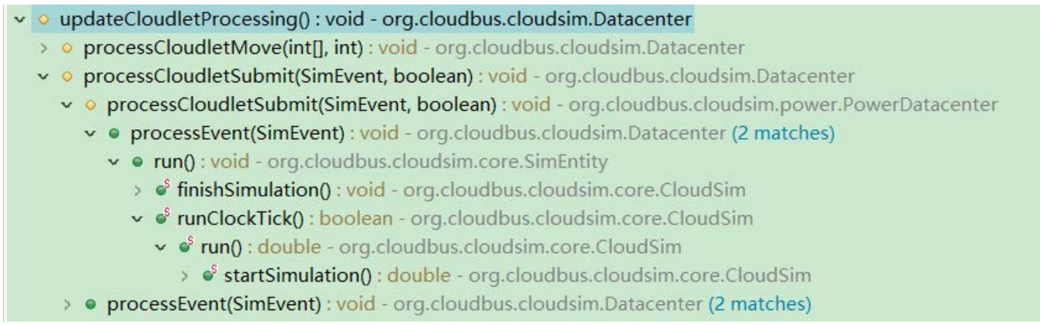
2. prob_cloud_selection: The probability that we choose the cloud to execute the applications.
3. active_period: In active period, the edge devices can generate new applications.
4. idle_period: In idle period, the edge devices can not generate new applications.

A sample of new application.xml shows as follows:

```
<?xml version="1.0"?>
<applications>
    <application name="TASK_BASED_APP">
        <usage_percentage>100</usage_percentage>
        <prob_cloud_selection>20</prob_cloud_selection>
        <poisson_interarrival>2</poisson_interarrival>
        <active_period>40</active_period>
        <idle_period>20</idle_period>
        <data_upload>6000</data_upload>
        <data_download>1000</data_download>
        <application_length>48000</application_length>
        <required_core>1</required_core>
        <vm_utilization_on_edge>8</vm_utilization_on_edge>
        <vm_utilization_on_cloud>0.8</vm_utilization_on_cloud>
        <vm_utilization_on_mobile>20</vm_utilization_on_mobile>
        <kernels>
            <kernel name="kernel1">
                <usage_percentage>100</usage_percentage>
                <prob_cloud_selection>20</prob_cloud_selection>
                <poisson_interarrival>2</poisson_interarrival>
                <active_period>40</active_period>
                <idle_period>20</idle_period>
                <data_upload>1500</data_upload>
                <data_download>250</data_download>
                <kernel_length>12000</kernel_length>
                <required_core>1</required_core>
                <vm_utilization_on_edge>8</vm_utilization_on_edge>
                <vm_utilization_on_cloud>0.8</vm_utilization_on_cloud>
                <vm_utilization_on_mobile>20</vm_utilization_on_mobile>
                <sub_application_number>0</sub_application_number>
                <num_dependency>0</num_dependency>
            </kernel>
            <kernel name=”kernel2”>
                <usage_percentage>100</usage_percentage>
                <prob_cloud_selection>20</prob_cloud_selection>
                <poisson_interarrival>2</poisson_interarrival>
                <active_period>40</active_period>
                <idle_period>20</idle_period>
                <data_upload>1500</data_upload>
                <data_download>250</data_download>
                <kernel_length>12000</kernel_length>
                <required_core>1</required_core>
                <vm_utilization_on_edge>8</vm_utilization_on_edge>
                <vm_utilization_on_cloud>0.8</vm_utilization_on_cloud>
                <vm_utilization_on_mobile>20</vm_utilization_on_mobile>
                <sub_application_number>1</sub_application_number>
                <num_dependency>0</num_dependency>
                <dependency>0</dependency>
            </kernel>
        </kernels>
    </application>
</applications>
```

The implementation of kernel-based application in EdgeCloudSim is developed by the abstraction of kernels. The original atomic task in EdgeCloudSim are changed to a kernel. An application can have one or multiple kernel. The kernel id is unique in this simulation tool. When an kernel-based application that have multiple kernels is created, an KernelBasedApplication.java class is instanciated and stores the kernel id of all the kernels that belongs to it. Moreover, this class handles the dependencies between the kernels in this application in runtime. All the instance of KernelBasedApplication is stored in a singleton instance of KernelBasedApplicationStatus.java. When we want to query or change the states of kernel-based application, we use the function in the singleton instance of KernelBasedApplicationStatus.java.

Adding the events of submitting kernels in runtime is required because we choose only submit a kernel to virtual machine when its dependencies has been met. This requirement is implemented using the event-driven simulation mechanism in CloudSim. In the CloudSim, so When a Cloudlet(kernel)



```
updateCloudletProcessing() : void - org.cloudbus.cloudsim.Datacenter
  processCloudletMove(int[], int) : void - org.cloudbus.cloudsim.Datacenter
  processCloudletSubmit(SimEvent, boolean) : void - org.cloudbus.cloudsim.Datacenter
    processCloudletSubmit(SimEvent, boolean) : void - org.cloudbus.cloudsim.power.PowerDatacenter
      processEvent(SimEvent) : void - org.cloudbus.cloudsim.Datacenter (2 matches)
        run() : void - org.cloudbus.cloudsim.core.SimEntity
          finishSimulation() : void - org.cloudbus.cloudsim.core.CloudSim
          runClockTick() : boolean - org.cloudbus.cloudsim.core.CloudSim
            run() : double - org.cloudbus.cloudsim.core.CloudSim
              startSimulation() : double - org.cloudbus.cloudsim.core.CloudSim
    processEvent(SimEvent) : void - org.cloudbus.cloudsim.Datacenter (2 matches)
```

Figure 2: Call hierarchy of function updateCloudletProcessing.

is finished, it will send a CLOUDLET_RETURN event to DatacenterBroker. In EdgeCloudSim, this event will be sent to MobileDeviceManager. Then processCloudletReturn function will be triggered. So when a kernel is finished, the kernels belongs to the same kernel-based applications will be checked. If there are kernels whose dependencies has been met now, it will be submitted to the VM at this time. The submit of kernel are based on the schedule function. We implement these functions in MobileDeviceManager class. What the scheduling function does is actually creating events for the entities. The entities in EdgeCloudsim are listed as follows:

1. SimManager: public class SimManager extends SimEntity
2. MobileDeviceManager: public abstract class MobileDeviceManager extends DatacenterBroker
3. EdgeOrchestrator: public abstract class EdgeOrchestrator extends SimEntity

Although we can access the kernel ID of kernels that needs to be submitted, the MobileDeviceManager cannot have access to the load generator. So send an event of submitting these kernels to SimManager. The following code shows the process that sends the event to SimManager in MobileDeviceManager.

```
if (KernelBasedApplicationStatus.getInstance().checkKernelInKBApp(kernelId))
    ↪ {
    int simManagerId =
        ↪ KernelBasedApplicationStatus.getInstance().getSimManagerId();
    // in the meanwhile set the kernel as finished
    List<Integer> kernelListReadySubmit =
         ↪ KernelBasedApplicationStatus.getInstance().getKernelSubmit(kernelId);
    for (int i=0; i < kernelListReadySubmit.size(); i++) {
        scheduleNow(simManagerId, 5, kernelListReadySubmit.get(i));
    }

}
```

The following code shows how the kernels is submitted in SimManager. Since the kernel needs to be allocated to specific virtual machines in a chosen host in runtime, the kernel is instanciated at this time. First we will retrieve KernelProperty instance related to this kernel, then pass it into the function submitKernel in class MobileDeviceManager. Actually now goes back to MobileDeviceManager instance.

```
case CREATE_KERNEL_IN_KBAPP:
    int kernelId = (int) ev.getData();
    //boolean ready_to_submit =
         ↪ KernelBasedApplicationStatus.getInstance().checkReadySubmit(kernelId);
    //get the kernel list index in load generator
    int kernelPropertyIndex =
         ↪ loadGeneratorModel.getKernelPropertyIndex(kernelId);
    KernelProperty kernelProperty =
         ↪ loadGeneratorModel.getKernelPropertyList().get(kernelPropertyIndex);
    mobileDeviceManager.submitKernel(kernelProperty);
    KernelBasedApplicationStatus.getInstance().setKernelSubmit(kernelId);
```

The following code shows a small part of the submitKernel function in MobileDeviceManager.

```
public void submitKernel(KernelProperty edgeTask) {
    NetworkModel networkModel =
         ↪ SimManager.getInstance().getNetworkModel();

    //create a task
    Kernel task = createTask(edgeTask);

    Location currentLocation = SimManager.getInstance().getMobilityModel().
            getLocation(task.getMobileDeviceId(),CloudSim.clock());
    //set location of the mobile device which generates this task
    task.setSubmittedLocation(currentLocation);

    //add related task to log list
    SimLogger.getInstance().addLog(task.getCloudletId(),
            task.getKernelType(),
            (int)task.getCloudletLength(),
            (int)task.getCloudletFileSize(),
            (int)task.getCloudletOutputSize(),
            (int)task.getKernelId());

    int nextHopId =
        ↪ SimManager.getInstance().getEdgeOrchestrator().getDeviceToOffload(task);
    ......
    ......
}
```

3.2 Allocation of Kernel-Based Application

The allocation of kernel-based application is based on the following steps. First, the parameters of different kinds of applications is loaded into SimSettings.java and stored in the following variables:

1. private double[][] applicationLookUpTable: store the parameters of applications.

2. private double[][] kernelInKernelBasedAPPLookUpTable: store the parameters of kernels in kernel-based application

Then load generator will generate all the applications and set their start time. The following code only contains some important lines from LoadGenerator. Singleton KernelBasedApplicationStatus stores all the runtime information of the kernel-based applications. All LernelBasedApplication instance are stored in KernelBasedApplicationStatus. At this time, we only create the kernel ID and create an kernelProperty instance for each kernel to store its information. An instance of Kernel.java will be created later and the runtime information will be stored in the kernel instance. Moreover, a kernelPropertyList is created and stores the proeprty of al the kernels.

```
while(virtualTime < simulationTime) {
    ......
    // check whether is kernel−based application that have multiple kernels
    if
         ↪ (SimSettings.getInstance().isKernelBasedApplication(randomApplicationType))
         ↪ {
        // create instance of kernel−based application
        // add the instance to KernelBasedapplication
        KernelBasedApplicationStatus.getInstance().addKernelBasedApplication(kernelNum,
             ↪ kernelBasedAppId);
        // add all the kernelProperty in this application to kernelPropertyList
    }
    else {
        atomicAppNum++;
        kernelPropertyList.add(new KernelProperty(i,randomApplicationType,
             ↪ virtualTime, expRngList, kernelId));
        kernelId++;
    }
}
```

When simulation starts, SimManager will loop over kernelPropertyList. It will check whether the kernel belongs to a kernel-based application that have multiple kernels. If it is, the kernels that dependencies have been met will be extracted. Then event of submitting these kernels at the start time of the applicaions that they belong to will be added for these kernels. If it is an application with a single kernel, an event of submitting this kenel at the start time of the corresponding applications will be added. The code are shown as follows.

```
//Creation of applications are scheduled here!
for(int i=0; i< loadGeneratorModel.getKernelPropertyList().size(); i++) {
    int taskPropertyId =
         ↪ loadGeneratorModel.getKernelPropertyList().get(i).getKernelId();
    if
         ↪ (KernelBasedApplicationStatus.getInstance().checkKernelInKBApp(taskPropertyId))
         ↪ {
        boolean ready_to_submit =
             ↪ KernelBasedApplicationStatus.getInstance().checkReadySubmit(taskPropertyId);
        if (ready_to_submit) {
            schedule(getId(),
               ↪ loadGeneratorModel.getKernelPropertyList().get(i).getStartTime(),
               ↪ CREATE_KERNEL_IN_ATOMIC,
               ↪ loadGeneratorModel.getKernelPropertyList().get(i));
            //TaskBasedTaskStatus.getInstance().setTaskSubmit(taskPropertyId);
            // cannot set the sub−task as submitted here, otherwise, all the task
               ↪ will be submitted
            KernelBasedApplicationStatus.getInstance().setKernelSubmit(taskPropertyId);
        }
    }
    else {
        schedule(getId(),
           ↪ loadGeneratorModel.getKernelPropertyList().get(i).getStartTime(),
           ↪ CREATE_KERNEL_IN_ATOMIC,
           ↪ loadGeneratorModel.getKernelPropertyList().get(i));
    }
}
```

When some kenels in kernel-based applications ends, there may be some kernels that can be submitted. The dependencies between kernels are maintained in KernelBasedApplicationStatus. When MobileDeviceManager knows some kernels ends. It will check whether the task is a kernel in kernel-based applications that have multiple kernels. If it is, Then it will check whether there are other kernels can be submitted after the ending of this kernel. If there are such kernels, the kernels will sbumitted. How to submit these kernels are described at the end of section 3.1.

3.3 Mapping between Kernel-Based Application and it's kernel

3.4 Classes Implementation

3.4.1 KernelProperty.java

KernelProperty.java are changed from the taskProperty.java in EdgeCloudSim. The kernelProperty.java are instantiated in load generator module. And the kernel ID are also generated in load generator and stored in KernelProperty.java.The fields in KernelProperty.java are listed as follows:

```
private double startTime;
private long length, inputFileSize, outputFileSize;
private int applicationType;
private int pesNumber;
private int mobileDeviceId;
private int kernelId;
```

3.4.2 Kernel.java

Kernel.java are changed from Task.java in EdgeCloudSim. Kernel.java extends Cloudlet.java class from CloudSim. In CloudSim, Cloudlet is the task scheduled and executed. Kernel will be submitted to CloudSim base. KernelProperty.java are instantiated using KernelProperty.java. Some fields in Kernel.java are determined in run-time. For example, mobileDeviceId, hostIndex, vmIndex, datacenterId are determined in runtime using the scheduling and dispatching algorithm.

```
private int type;
private int mobileDeviceId;
private int hostIndex;
private int vmIndex;
private int datacenterId;
private int kernelId;
private Location submittedLocation;
```

3.4.3 KernelBasedApplicationSettings.java

KernelBasedApplicationSettings.java is used to store the task dependency. The following code shows the fields in KernelBasedApplicationSettings.java. Table 1 shows an example of dependency in KernelBasedApplicationSettings.java. If kernel 1 depends on kernel 2, then the value whose row is 1 and column is 2 is 1. Otherwise, the value in the table is 0. In the example given by the table, kernel 1 and kernel 2 can start at the start time of the application. Kernel 3 can only start after the end of kernel 1. Kernel 4 can only start after the end of kernel 1 and kernel 2.

```
private int startIndex;
private int numKernel;
private int[][] dependency
```

0	0	0	0
0	0	0	0
1	0	0	0
1	1	0	0

Table 1: An example of dependency in KernelBasedApplicationSettings.java.

3.4.4 KernelBasedApplication.java

This class is created in load generator. When we initialize the simulation, all the application that will be created in the simulation is determined and this class is instanciated. This class is used to store the runtime status of the kernel-based application. Some important fields of this class are shown as follows:

```java
public class KernelBasedApplication {
    private int numKernel;
    // map from the KernelId to the inner index id
    private Map<Integer, Integer>kernelId2Index;
    // map from the inner index id to KernelId
    private Map<Integer, Integer>index2KernelId;
    private boolean[] submitted;
    // finished used to check if the task has ended
    private boolean[] finished;
    // store the final status of the execution of all kernels in the
        ↪ KernelBasedApplication
    // −1: not tracked 0: finished 1: not finished 2: failed
    private int kbAPPFinalStatus;
    /*
     * dependency[1][0] = 1 means kernel 1 can only be executed after on kernel
         ↪ 0 is ended
     *  0 1
     * 0
     * 1
     */
    private int[][] dependency;
    /*
     * Stores the run−time kernel−dependency graph information
     * initial: if dependency[0][1] = 1, dependency_met[0][1] = 0
     *    otherwise, dependency_met[0][1] = 1
     * dependency_met[0][1] = 1 means dependency for 0 is met considering 1
     */
    private int[][] dependency_met;
    ......
}
```

3.4.5 KernelBasedApplicationStatus.java

This class is a singleton and stores all the instance of KernelBasedApplication.java. Each module can access this class using the static function getInstance in KernelBasedApplicationStatus.java class. Some important fields of this class are shown as follows:

```
public class KernelBasedApplicationStatus {
   private static KernelBasedApplicationStatus instance = null;
   // used when we want to add an event to SimManager
   private int simManagerId;
   private int numKernelBasedApplication;
   private Map<Integer, KernelBasedApplication> kernelBasedApplicationMap;
   // map kernelBasedApplication id to the key in kernelBasedApplicationMap
   private Map<Integer, Integer> mKeyMap;

   public static KernelBasedApplicationStatus getInstance() {
      if (instance == null) {
         instance = new KernelBasedApplicationStatus();
      }
      return instance;
   }
   ...
}
```

3.4.6 SimLogger.java and LogItem.java

A new field kernelId is added to the LogItem.java class to map map the kernel to the log LogItem.

```
class LogItem {
   private int kernelId;
   ......
}
```

3.5 Simulation Workflow

The simulation works in the following steps. First SimManager will use the class SimSettings to load the settings about the application types, edge server, and some other parameters.

The generation of applications are in two phase. In the first phase, SimManager will utilize LoadGenerator to generate the applications that will be scheduled in the simulation. In this process, LoadGenerator will use Possion process to get the arrival time of each applications and use the parameters in SimSettings to generate the positions of each applicaion. Then comes to the second phase, it's a loop to submit the kernels in the application to be scheduled. The loop ends when the simulation comes to the end of the simulation time. In each loop, SimManager will check its the events that needs to be processed at that time. Before the start of the loop, there are two types of events that are determined. The first type is the kernel in applications that only have one kernel. Their start time are determined so the submitting those kernel at the start time of the kernel-based application is added to the events of SimManager. The other type of events is related to the kernels of kernel-based application. The events of submitting kernels that do not have any dependencies in kernel-based applications will be added. When some kernels finishes, some other kernels that depends on it can be submitted at that time. The events of submitting these kernels will be added at run time.

After the kernel is submitted, it is scheduled in CloudSim. Figure shows the Cloudlet processing update process in CloudSim. The original in the paper about CloudSim is wrong. Figure 2 is the fixed figure about Cloudlet processing update process. EdgeCloudSim will decide the device(DataCenter in CloudSim), host, and virtual machine to submit the kernel in runtime. After submitting the kernel, the schedulling process in the virtual machines is handled by CloudSim.

4 Experimental Results

In this section, we did some experiment. We test the failure rate.

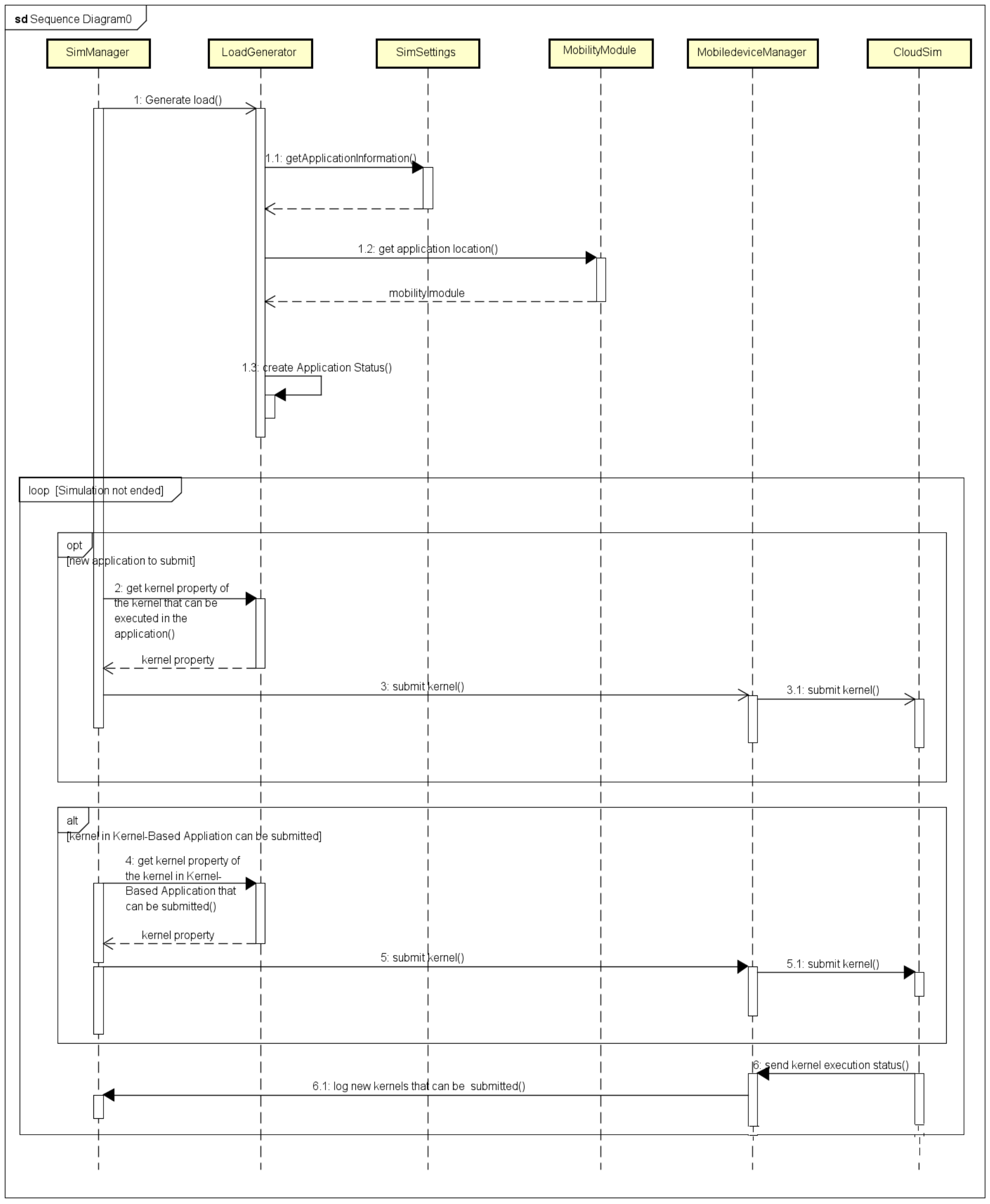

sd Sequence Diagram0
SimManager
LoadGenerator
SimSettings
MobilityModule
MobiledeviceManager
CloudSim
1: Generate load()
1.1: getApplicationInformation()
1.2: get application location()
mobility module
1.3: create Application Status()
loop [Simulation not ended]
opt
[new application to submit]
2: get kernel property of the kernel that can be executed in the application()
kernel property
3: submit kernel()
3.1: submit kernel()
alt
[kernel in Kernel-Based Appliation can be submitted]
4: get kernel property of the kernel in Kernel-Based Application that can be submitted()
kernel property
5: submit kernel()
5.1: submit kernel()
6: send kernel execution status()
6.1: log new kernels that can be submitted()

Figure 3: Modules and functions related to create and submit kernel.

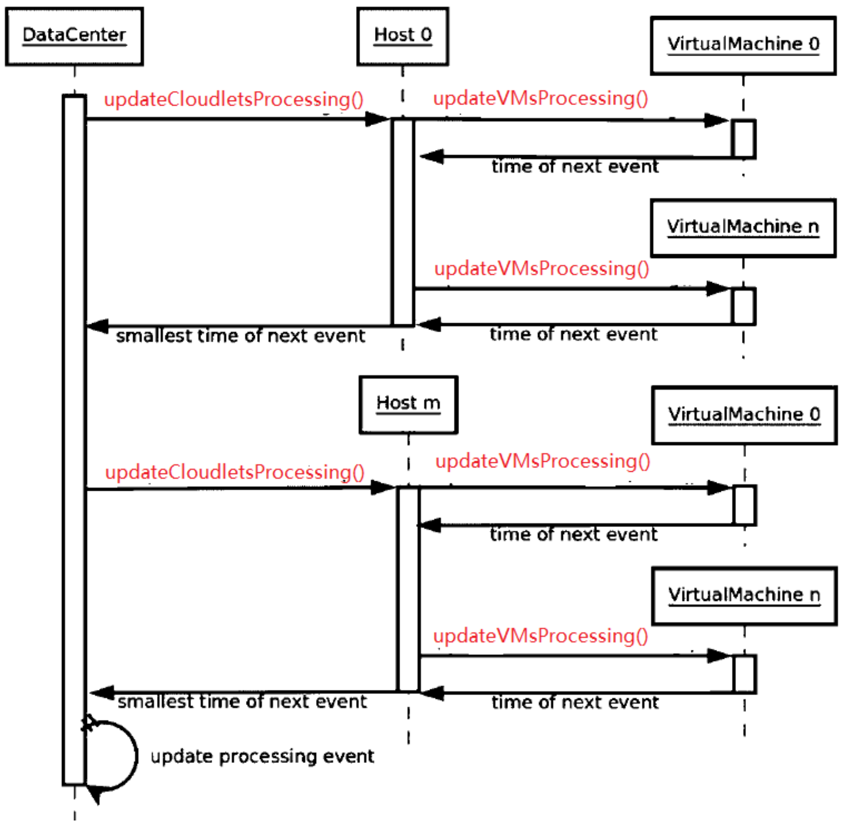


Figure 4: Cloudlet processing update process.

5 Conclusions

6 Future Works