

VISVESVARAYA TECHNOLOGICAL UNIVERSITY, BELAGAVI 590018



Report on

**“SIMULATION OF MULTI LEVEL PRIORITY QUEUE FOR TASK
SCHEDULING IN CLOUD-BASED ENVIRONMENT”**

By

DEEPTHI BHAT (1BM16CS003)

AISHWARYA N (1BM16CS010)

ADITI AWASTHI (1BM16CS008)

Under the Guidance of

RAJESHWARI B S

ASSISTANT PROFESSOR, Department of CSE

BMS College of Engineering

Work carried out at



Department of Computer Science and Engineering

BMS College of Engineering

(Autonomous college under VTU)

P.O. Box No.: 1908, Bull Temple Road, Bangalore-560 019

2018-2019

BMS COLLEGE OF ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING



CERTIFICATE

This is to certify that the Cloud Computing assignment titled “SIMULATION OF MULTI LEVEL PRIORITY QUEUE FOR TASK SCHEDULING IN CLOUD-BASED ENVIRONMENT” has been carried out by DEEPTHI BHAT (1BM16CS003), AISHWARYA N (1BM16CS010) and ADITI AWASTHI (1BM16CS008) during the academic year 2018-2019.

Signature of the guide
RAJESHWARI B S
ASSISTANT PROFESSOR,
Department of Computer Science and Engineering
BMS College of Engineering, Bangalore

BMS COLLEGE OF ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING



DECLARATION

We, DEEPTHI BHAT (1BM16CS003), AISHWARYA N (1BM16CS010) and ADITI AWASTHI (1BM16CS008), students of 6th Semester, B.E, Department of Computer Science and Engineering, BMS College of Engineering, Bangalore, hereby declare that, this assignment work entitled " SIMULATION OF MULTI LEVEL PRIORITY QUEUE FOR TASK SCHEDULING IN CLOUD-BASED ENVIRONMENT " has been carried out by us under the guidance of **RAJESHWARI B S**, ASSISTANT PROFESSOR, Department of CSE, BMS College of Engineering, Bangalore during the academic semester Jan-May 2019. We also declare that to the best of our knowledge and belief, the assignment reported here is not from part of any other report by any other students.

Signature of the Candidates

DEEPTHI BHAT (1BM16CS003)

AISHWARYA N (1BM16CS010)

ADITI AWASTHI (1BM16CS008)

1. OBJECTIVES

1.1 To implement Proportional Scheduling, Priority Based Scheduling, Shortest Job First and First Come First Serve algorithms for processes or tasks residing on different levels of a Multi-Level Priority Queue.

1.2 To plot graphs showing the following:-

- a) Waiting Time of tasks assigned to VMs.
- b) Response Time of tasks.
- c) Resource Utilization of VMs.
- d) Number of cloudlets run on each VM.

2. INTRODUCTION

Recently, cloud computing emerged as the leading technology for delivering reliable, secure, fault-tolerant, sustainable, and scalable computational services, which are presented as Software, Infrastructure, or Platform as services (SaaS, IaaS, PaaS). Moreover, these services may be offered in private data centers (private clouds), may be commercially offered for clients (public clouds), or yet it is possible that both public and private clouds are combined in hybrid clouds.

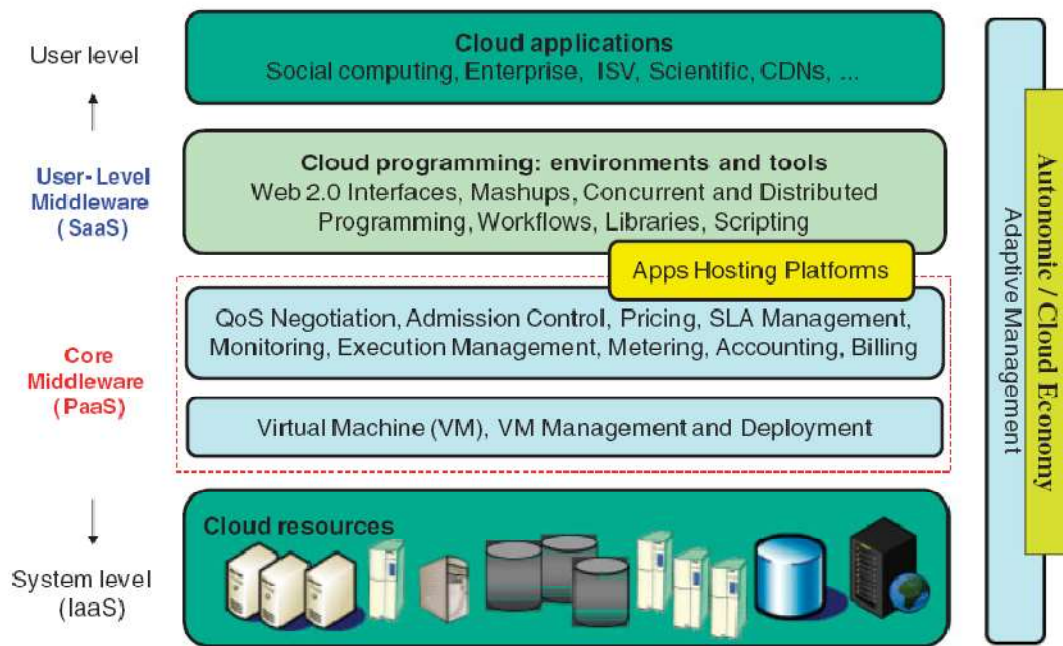


Figure 1

Above diagram shows the layered design of Cloud computing architecture. Physical Cloud resources along with core middleware capabilities form the basis for delivering IaaS and PaaS. The user-level middleware aims at providing SaaS capabilities. The top layer focuses on application services (SaaS) by making use of services provided by the lower-layer services. PaaS/SaaS services are often developed and provided by third-party service providers, who are different from the IaaS providers.

Cloud applications: This layer includes applications that are directly available to end-users. We define end-users as the active entity that utilizes the SaaS applications over the Internet. These applications may be supplied by the Cloud provider (SaaS providers) and accessed by end-users either via a subscription model or on a pay-per-use basis. Alternatively, in this layer, users deploy their own applications. In the former case, there are applications such as Salesforce.com that supply business process models on clouds (namely, customer relationship management software) and social networks. In the latter, there are e-Science and e-Research applications, and Content-Delivery Networks.

User-Level middleware: This layer includes the software frameworks, such as Web 2.0 Interfaces (Ajax, IBM Workplace), that help developers in creating rich, cost-effective user-interfaces for browser-based applications. The layer also provides those programming environments and composition tools that ease the creation, deployment, and execution of applications in clouds. Finally, in this layer several frameworks that support multi-layer applications development, such as Spring and Hibernate, can be deployed to support applications running in the upper level

Core middleware: This layer implements the platform-level services that provide run-time environment for hosting and managing User-Level application services. The core services at this layer include Dynamic SLA Management, Accounting, Billing, Execution monitoring and management, and Pricing (are all the services to be capitalized?). The well-known examples of services operating at this layer are Amazon EC2, Google App Engine, and Aneka. The functionalities exposed by this layer are accessed by both SaaS (the services represented at the top-most layer in above diagram) and IaaS (services shown at the bottom-most layer in above diagram) services. Critical functionalities that need to be realized at this layer include messaging, service discovery, and load-balancing. These functionalities are usually implemented by Cloud providers and offered to application developers at an additional premium. For instance, Amazon offers a load-balancer and a monitoring service (Cloudwatch) for the Amazon EC2 developers/consumers. Similarly, developers building applications on Microsoft Azure clouds can use the .NET Service Bus for implementing message passing mechanism.

System Level: The computing power in Cloud environments is supplied by a collection of data centers that are typically installed with hundreds to thousands of hosts. At the System-Level layer, there exist massive physical resources (storage servers and application servers) that power the data centers. These servers are transparently managed by the higher-level virtualization services and toolkits that allow sharing of their capacity among virtual instances of servers. These VMs are isolated from each other, thereby making fault tolerant behavior and isolated security context possible.

These already wide ecosystem of cloud architectures, along with the increasing demand for energy-efficient IT technologies, demand timely, repeatable, and controllable methodologies for evaluation of algorithms, applications, and policies before actual development of cloud products. Because utilization of real testbeds limits the experiments to the scale of the testbed and makes the reproduction of results an extremely difficult undertaking, alternative approaches for testing and experimentation leverage development of new Cloud technologies.

A suitable alternative is the utilization of simulations tools, which open the possibility of evaluating the hypothesis prior to software development in an environment where one can reproduce tests. Specifically in the case of Cloud computing, where access to the infrastructure incurs payments in real currency, simulation-based approaches offer significant benefits, as it allows Cloud customers to test their services in repeatable and controllable environment free of cost, and to tune the performance bottlenecks before deploying on real Clouds. At the provider side, simulation environments allow evaluation of different kinds of resource leasing scenarios under varying load and pricing distributions. Such studies could aid the providers in optimizing the resource access cost with focus on improving profits. In the absence of such simulation platforms, Cloud customers and providers have to rely either on theoretical and imprecise

evaluations, or on try-and-error approaches that lead to inefficient service performance and revenue generation.

The primary objective of this project is to provide a generalized, and extensible simulation framework that enables seamless modeling, simulation, and experimentation of emerging Cloud computing infrastructures and application services. By using CloudSim, researchers and industry-based developers can focus on specific system design issues that they want to investigate, without getting concerned about the low level details related to Cloud-based infrastructures and services.

2.1 Main features

Overview of CloudSim functionalities:

- support for modeling and simulation of large scale Cloud computing data centers
- support for modeling and simulation of virtualized server hosts, with customizable policies for provisioning host resources to virtual machines
- support for modeling and simulation of application containers
- support for modeling and simulation of energy-aware computational resources
- support for modeling and simulation of data center network topologies and message-passing applications
- support for modeling and simulation of federated clouds
- support for dynamic insertion of simulation elements, stop and resume of simulation
- support for user-defined policies for allocation of hosts to virtual machines and policies for allocation of host resources to virtual machines

3. Architecture

Figure 2 shows the multi-layered design of the CloudSim software framework and its Architectural components. Initial releases of CloudSim used SimJava as the discrete event simulation engine that supports several core functionalities, such as queuing and processing of events, creation of Cloud system entities (services, host, data center, broker, VMs), communication between components and management of the simulation clock.

The CloudSim simulation layer provides support for modeling and simulation of virtualized Cloud-based data center environments including dedicated management interfaces for VMs, memory, storage, and bandwidth. The fundamental issues, such as provisioning of hosts to VMs, managing application execution, and monitoring dynamic system state, are handled by this layer. A Cloud provider, who wants to study the efficiency of different policies in allocating its hosts to VMs (VM provisioning), would need to implement his strategies at this layer. Such implementation can be done by programmatically extending the core VM provisioning functionality. There is a clear distinction at this layer related to provisioning of hosts to VMs.

A Cloud host can be concurrently allocated to a set of VMs that execute applications based on SaaS provider's defined QoS levels. This layer also exposes the functionalities that a Cloud application developer can extend to perform complex workload profiling and application performance study. The top-most layer in the CloudSim stack is the User Code that exposes basic entities for hosts (number of machines, their specification, and so on), applications (number of tasks and their requirements), VMs, number of users and their application types, and broker scheduling policies. By extending the basic entities given at this layer, a Cloud application developer can perform the following activities:

- generate a mix of workload request distributions, application configurations;
- model Cloud availability scenarios and perform robust tests based on the custom configurations; and
- implement custom application provisioning techniques for clouds and their federation.

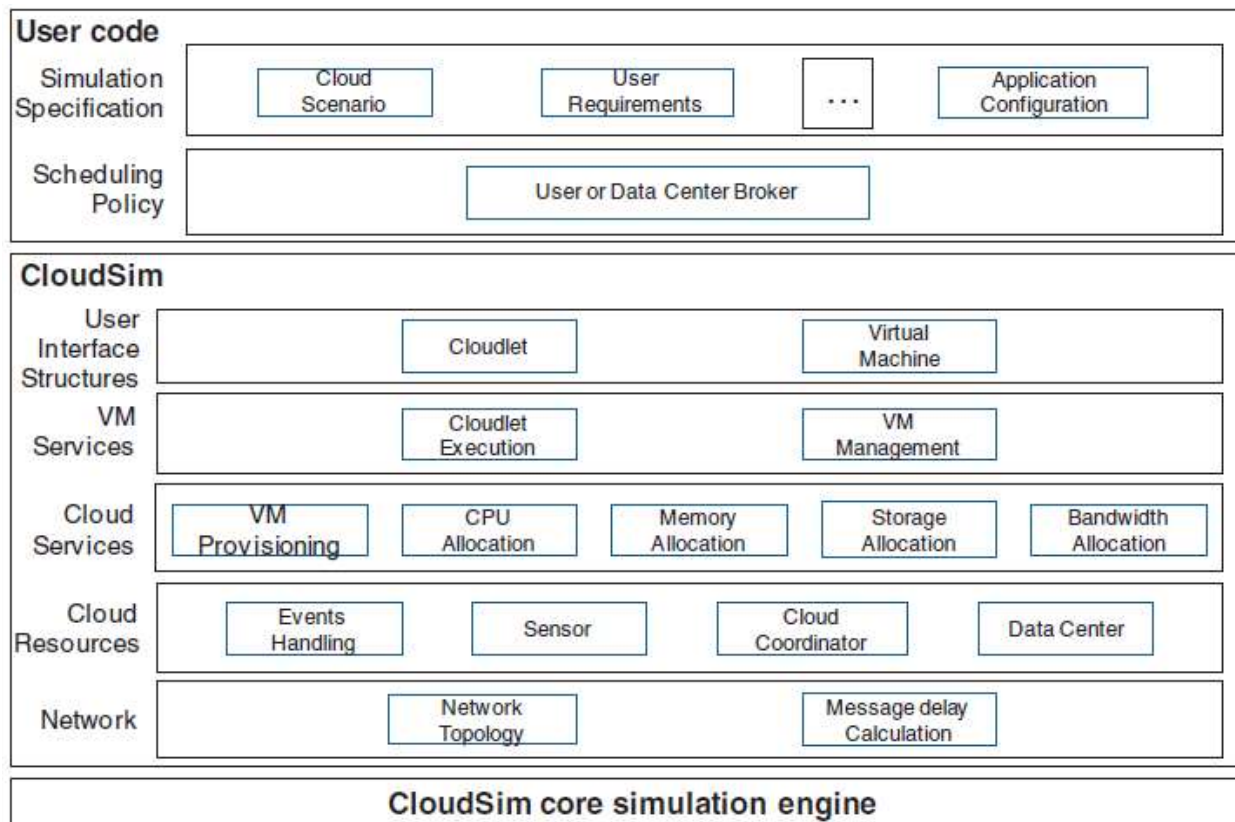


Figure 2

3.1 Modeling the cloud

The infrastructure-level services (IaaS) related to the clouds can be simulated by extending the datacenter entity of CloudSim. The data center entity manages a number of host entities. The hosts are assigned to one or more VMs based on a VM allocation policy that should be defined by the Cloud service provider. Here, the VM policy stands for the operations control policies related to VM life cycle such as: provisioning of a host to a VM, VM creation, VM destruction, and VM migration. Similarly, one or more application services can be provisioned within a single VM instance, referred to as application provisioning in the context of Cloud computing. In the context of CloudSim, an entity is an instance of a component. A CloudSim component can be a class (abstract or complete) or set of classes that represent one CloudSim model (data center, host).

A data center can manage several hosts that in turn manages VMs during their life cycles. Host is a CloudSim component that represents a physical computing server in a Cloud: it is assigned a pre-configured processing capability (expressed in millions of instructions per second—MIPS), memory, storage, and a provisioning policy for allocating processing cores to VMs. The Host component implements interfaces that support modelling and simulation of both single-core and multi-core nodes.

VM allocation (provisioning) is the process of creating VM instances on hosts that match the critical characteristics (storage, memory), configurations (software environment), and requirements (availability zone) of the SaaS provider.

CloudSim supports the development of custom application service models that can be deployed within a VM instance and its users are required to extend the core Cloudlet object for implementing their application services. Furthermore, CloudSim does not enforce any limitation on the service models or provisioning techniques that developers want to implement and perform tests with. Once an application service is defined and modeled, it is assigned to one or more pre-instantiated VMs through a service-specific allocation policy.

Allocation of application-specific VMs to hosts in a Cloud-based data center is the responsibility of a VM Allocation controller component (called `VmAllocationPolicy`). This component exposes a number of custom methods for researchers and developers who aid in the implementation of new policies based on optimization goals (user centric, system centric, or both). By default, `VmAllocationPolicy` implements a straightforward policy that allocates VMs to the Host on a First-Come-First-Serve (FCFS) basis. Hardware requirements, such as the number of processing cores, memory, and storage, form the basis for such provisioning. Other policies, including the ones likely to be expressed by Cloud providers, can also be easily simulated and modeled in CloudSim.

For each Host component, the allocation of processing cores to VMs is done based on a host allocation policy. This policy takes into account several hardware characteristics, such as number of CPU cores, CPU share, and amount of memory (physical and secondary), that are allocated to a given VM instance. Hence, CloudSim supports simulation scenarios that assign specific CPU cores to specific VMs (a space-shared policy), dynamically distribute the capacity of a core among VMs (time-shared policy), or assign cores to VMs on demand.

Each host component also instantiates a VM scheduler component, which can either implement the space-shared or the time-shared policy for allocating cores to VMs. Cloud system/application developers and researchers can further extend the VM scheduler component for experimenting with custom allocation policies. In the next section, the finer-level details related to the time-shared and space-shared policies are described.

Fundamental software and hardware configuration parameters related to VMs are defined in the VM class. Currently, it supports modeling of several VM configurations offered by Cloud providers such as the Amazon EC2.

3.2 Modeling the VM allocation

One of the key aspects that make a Cloud computing infrastructure different from a Grid computing infrastructure is the massive deployment of virtualization tools and technologies. Hence, as against Grids, Clouds contain an extra layer (the virtualization layer) that acts as an execution, management, and hosting environment for application services. Hence, traditional application provisioning models that assign individual application elements to computing nodes do not accurately represent the computational abstraction, which is commonly associated with Cloud resources.

For example, consider a Cloud host that has a single processing core. There is a requirement of concurrently instantiating two VMs on that host. Although in practice VMs are contextually (physical and secondary memory space) isolated, still they need to share the processing cores and system bus. Hence, the amount of hardware resources available to each VM is constrained by the total processing power and system bandwidth available within the host. This critical factor must be considered during the VM provisioning process, to avoid creation of a VM that demands more processing power than is available within the host. In order to allow simulation of different provisioning policies under varying levels of performance isolation, CloudSim supports VM provisioning at two levels: first, at the host level and second, at the VM level. At the host level, it is possible to specify how much of the overall processing power of each core will be assigned to each VM.

At the VM level, the VM assigns a fixed amount of the available processing power to the individual application services (task units) that are hosted within its execution engine. We consider a task unit as a finer abstraction of an application service being hosted in the VM.

At each level, CloudSim implements the time-shared and space-shared provisioning policies. To clearly illustrate the difference between these policies and their effect on the application service performance, in Figure 4 we show a simple VM provisioning scenario. In this figure, a host with two CPU cores receives request for hosting two VMs, such that each one requires two cores and plans to host four tasks' units. More specifically, tasks t_1, t_2, t_3 , and t_4 to be hosted in VM1, whereas t_5, t_6, t_7 , and t_8 to be hosted in VM2.

Figure 3(a) presents a provisioning scenario, where the space-shared policy is applied to both VMs and task units. As each VM requires two cores, in space-shared mode only one VM can run at a given instance of time. Therefore, VM2 can only be assigned the core once VM1 finishes the execution of task units. The same happens for provisioning tasks within the VM1: since each task unit demands only one core, therefore both of them can run simultaneously. During this period, the remaining tasks (2 and 3) wait in the execution queue. By using a space-shared policy, the estimated finish time of a task p managed by a VM i is given by

$$eft(p) = est + \frac{rl}{capacity \times cores(p)},$$

Where $est(p)$ is the Cloudlet- (cloud task) estimated start time and rl is the total number of instructions that the Cloudlet will need to execute on a processor. The estimated start time

dependson the position of the Cloudlet in the execution queue, because the processing unit is usedexclusively (space-shared mode) by the Cloudlet. Cloudlets are put in the queue when there arefree processing cores available that can be assigned to the VM. In this policy, the total capacityof a host having np processing elements (PEs) is given by:

$$capacity = \sum_{i=1}^{np} \frac{cap(i)}{np},$$

Where $cap(i)$ is the processing strength of individual elements.

In Figure 3(b), a space-shared policy is applied for allocating VMs to hosts and a time-shared policy forms the basis for allocating task units to processing core within a VM. Hence, during a VM lifetime, all the tasks assigned to it are dynamically context switched during their life cycle. Byusing a time-shared policy, the estimated finish time of a Cloudlet managed by a VM is given by

$$eft(p) = ct + \frac{rl}{capacity \times cores(p)},$$

where $eft(p)$ is the estimated finish time, ct is the current simulation time, and $cores(p)$ is the number of cores (PEs) required by the Cloudlet. In time-shared mode, multiple Cloudlets (task units) can simultaneously multi-task within a VM. In this case, we compute the total processing capacity of Cloud host as

$$capacity = \frac{\sum_{i=1}^{np} cap(i)}{\max \left(\sum_{j=1}^{cloudlets} cores(j), np \right)},$$

where $cap(i)$ is the processing strength of individual elements.

In Figure 3(c), a time-shared provisioning is used for VMs, whereas task units are provisioned based on a space-shared policy. In this case, each VM receives a time slice on each processing core, which then distributes the slices among task units on a space-shared basis. As the cores are shared, the amount of processing power available to a VM is variable. This is determined by calculating VMs that are active on a host. As the task units are assigned based on a space-shared policy, which means that at any given instance of time only one task will be actively using the processing core.

Finally, in Figure 3(d) a time-shared allocation is applied to both VMs and task units. Hence, the processing power is concurrently shared by the VMs and the shares of each VM are simultaneouslydivided among its task units. In this case, there are no queuing delays associated with task units.

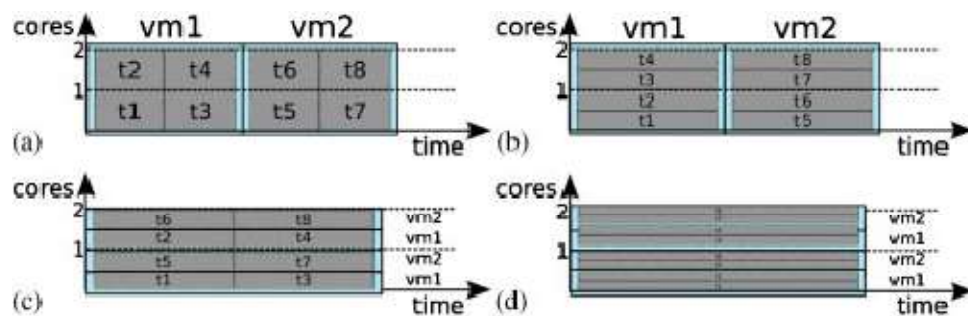


Figure 3

Effects of different provisioning policies on task unit execution: (a) space-shared provisioning for VMs and tasks; (b) space-shared provisioning for VMs and time-shared provisioning for tasks; (c) time-shared provisioning for VMs, space-shared provisioning for tasks; and (d) time-shared provisioning for VMs and tasks.

3. EXAMPLE PROGRAMS

DATA CENTER

- ❑ Datacenters are the resource providers in cloudsim.
- ❑ To create a Power Datacenter:
 - Create a list to store the machines.
`List<Host> hostList = new ArrayList<Host>();`
 - Each machine contains one or more PEs or CPUs/Cores.
`List<Pe> peList = new ArrayList<Pe>();`
 - Create PEs and add these into a list
`peList.add(new Pe(o, new PeProvisionerSimple(mips)));`
 - Create Host with its id and list of PEs and add them to the list of machines.

```
hostList.add(  
    new Host(hostId,  
    new RamProvisionerSimple(ram),  
    new BwProvisionerSimple(bw),  
    storage,  
    peList,  
    new VmSchedulerTimeShared(peList))  
);
```

- Create a Datacenter Characteristics object that stores the properties of a data center: architecture, OS, list of Machines, allocation policy: time- or space-shared, time zone and its price.
`DatacenterCharacteristics char = new DatacenterCharacteristics(arch, os, vmm, hostList, time_zone, cost, costPerMem, costPerStorage, costPerBw);`
- Create a Power Datacenter object.
`datacenter = new Datacenter(name, characteristics, new VmAllocationPolicySimple(hostList), storageList, 0);`

CloudSim Example 1: A simple example showing how to create a datacenter with one host and run one cloudlet on it.

- ❑ Initialize cloudsim package.
- ❑ Create datacentre:

```
Datacenter datacenter0 = createDatacenter("Datacenter_0");
```
- ❑ Create broker:

```
DatacenterBroker broker = createBroker();
```
- ❑ A virtual machine is created:

```
Vm vm = new Vm(vmid, brokerId, mips, pesNumber, ram, bw, size, vmm, new CloudletSchedulerTimeShared());
```
- ❑ Create a cloudlet list, the cloudlet and VM list are submitted to the broker.
- ❑ Start the simulation and print the results once the simulation is over.


CloudSim Example 2: A simple example showing how to create a datacenter with one host and run two cloudlets on it. The cloudlets run in VMs with the same MIPS requirements. The cloudlets will take the same time to complete the execution.

- ❑ Initialize cloudsim package.
- ❑ Create datacentre:

```
Datacenter datacenter0 = createDatacenter("Datacenter_0");
```
- ❑ Create broker:

```
DatacenterBroker broker = createBroker();
```
- ❑ Two virtual machines are created:


```
Vm vm1 = new Vm(vmid, brokerId, mips, pesNumber, ram, bw, size, vmm, new CloudletSchedulerTimeShared());  
vmid++;  
Vm vm2 = new Vm(vmid, brokerId, mips, pesNumber, ram, bw, size, vmm, new CloudletSchedulerTimeShared());
```


- 
- ❑ The VM list is submitted to the broker.
 - ❑ Similarly two cloudlets are created and bound to the VMs. In this manner, the broker will submit the bound cloudlets only to specific VMs.

```
Cloudlet cloudlet = new Cloudlet(id, length, pesNumber, fileSize,
                                outputSize, utilizationModel, utilizationModel,
                                utilizationModel);

cloudlet.setUserId(brokerId);
broker.bindCloudletToVm(cloudlet.getCloudletId(),vm.getId());
```
 - ❑ Start the simulation and print the results once the simulation is over.

CloudSim Example 3: A simple example showing how to create a datacenter with two hosts and run two cloudlets on it. The cloudlets run in VMs with different MIPS requirements. The cloudlets will take different time to complete the execution depending on the requested VM performance.

- 
- ❑ Initialize cloudsim package.
 - ❑ Create datacentre:

```
Datacenter datacenter0 = createDatacenter("Datacenter_0");
```
 - ❑ Create broker:

```
DatacenterBroker broker = createBroker();
```
 - ❑ Two virtual machines are created with different mips req:

```
Vm vm1 = new Vm(vmid, brokerId, mips, pesNumber, ram, bw,
                size, vmm, new CloudletSchedulerTimeShared());


vmid++;
Vm vm2 = new Vm(vmid, brokerId, mips * 2, pesNumber, ram,
                bw, size, vmm, new CloudletSchedulerTimeShared());
```


- 
- ❑ The VM list is submitted to the broker.
 - ❑ Similarly two cloudlets are created and bound to the VMs. In this manner, the broker will submit the bound cloudlets only to specific VMs.

```
Cloudlet cloudlet = new Cloudlet(id, length, pesNumber, fileSize,
                                outputSize, utilizationModel, utilizationModel,
                                utilizationModel);

cloudlet.setUserId(brokerId);
broker.bindCloudletToVm(cloudlet.getId(),vm.getId());
```
 - ❑ Start the simulation and print the results once the simulation is over.

CloudSim Example 4: A simple example showing how to create two datacenters with one host each and run two cloudlets on them.


- 
- ❑ Initialize cloudsim package.
 - ❑ Two datacenters are created:

```
Datacenter datacenter0 = createDatacenter("Datacenter_0");
Datacenter datacenter1 = createDatacenter("Datacenter_1");
```
 - ❑ Create broker:

```
DatacenterBroker broker = createBroker();
```
 - ❑ Two virtual machines are created:

```
Vm vm1 = new Vm(vmid, brokerId, mips, pesNumber, ram, bw,
                size, vmm, new CloudletSchedulerTimeShared());


vmid++;
Vm vm2 = new Vm(vmid, brokerId, mips, pesNumber, ram, bw,
                size, vmm, new CloudletSchedulerTimeShared());
```

- 
- ❑ The VM list is submitted to the broker.
 - ❑ Similarly two cloudlets are created and bound to the VMs. In this manner, the broker will submit the bound cloudlets only to specific VMs.

```
Cloudlet cloudlet = new Cloudlet(id, length, pesNumber, fileSize,
                                outputSize, utilizationModel, utilizationModel,
                                utilizationModel);

cloudlet.setUserId(brokerId);
broker.bindCloudletToVm(cloudlet.getId(),vm.getId());
```
 - ❑ Start the simulation and print the results once the simulation is over.

CloudSim Example 5: A simple example showing how to create two datacenters with one host each and run cloudlets of two users on them.

- 
- ❑ Initialize cloudsim package.
 - ❑ Two datacenters are created:

```
Datacenter datacenter0 = createDatacenter("Datacenter_0");
Datacenter datacenter1 = createDatacenter("Datacenter_1");
```
 - ❑ Two brokers are created:

```
DatacenterBroker broker1 = createBroker(1);
DatacenterBroker broker2 = createBroker(2);
```
 - ❑ One virtual machine is created for each broker/user where VM1 belongs to user1 and VM2 to user2.

```
Vm vm1 = new Vm(vmid, brokerId, mips, pesNumber, ram, bw,
                size, vmm, new CloudletSchedulerTimeShared());

Vm vm2 = new Vm(vmid, brokerId, mips, pesNumber, ram, bw,
                size, vmm, new CloudletSchedulerTimeShared());
```



- ❑ The two VMs are added to their respective lists and submitted to the broker.

```
broker1.submitVmList(vmlist1);  
broker2.submitVmList(vmlist2);
```

- ❑ Similarly two cloudlets are created. The two cloudlet lists are sent to the respective brokers.

```
Cloudlet cloudlet1 = new Cloudlet(id, length, pesNumber, fileSize,  
                                outputSize, utilizationModel, utilizationModel,  
                                utilizationModel);  
  
cloudlet.setUserId(brokerId1);  
broker1.submitCloudletList(cloudletList1);  
broker2.submitCloudletList(cloudletList2);
```

- ❑ Start the simulation and print the results once the simulation is over

CloudSim Example 6: An example showing how to create scalable simulations.



- ❑ Create a container to store VMs. This list is passed onto the broker later.

```
LinkedList<vm> list = new LinkedList<vm>();  
vm[i] = new Vm(i, userId, mips, pesNumber, ram, bw, size,  
              vmm, new CloudletSchedulerTimeShared());
```

- ❑ Create a container to store Cloudlets.

```
LinkedList<Cloudlet> list = new LinkedList<Cloudlet>();  
cloudlet[i] = new Cloudlet(i, length, pesNumber, fileSize,  
                          outputSize, utilizationModel, utilizationModel,  
                          utilizationModel);
```




- ❑ Initialize cloudsim package.
- ❑ Two datacenters are created:

```
Datacenter datacenter0 = createDatacenter("Datacenter_0");
Datacenter datacenter1 = createDatacenter("Datacenter_1");
```
- ❑ Create broker:

```
DatacenterBroker broker = createBroker();
```
- ❑ Create VMs and Cloudlets and send them to broker.

```
vmList = createVM(brokerId, 5, 0);
cloudletList = createCloudlet(brokerId, 10, 0);
```



- ❑ Create a thread that will create a new broker at 200 clock time.

```
Runnable monitor = new Runnable();
```

To pause simulation:

```
CloudSim.pauseSimulation(200);
```

To resume simulation:

```
CloudSim.resumeSimulation();
```
- ❑ Start the simulation and print the results once the simulation is over.

CloudSim Example 8: An example showing how to create simulation entities (a DatacenterBroker in this example) in run-time using a global manager entity (GlobalBroker).



- ❑ Create a container to store VMs. This list is passed onto the broker later.

```
LinkedList<Vm> list = new LinkedList<Vm>();  
vm[i] = new Vm(i, userId, mips, pesNumber, ram, bw, size,  
              vmm, new CloudletSchedulerTimeShared());
```

- ❑ Create a container to store Cloudlets.

```
LinkedList<Cloudlet> list = new LinkedList<Cloudlet>();  
cloudlet[i] = new Cloudlet(i, length, pesNumber, fileSize,  
                          outputSize, utilizationModel, utilizationModel,  
                          utilizationModel);
```



- ❑ Initialize cloudsim package.

- ❑ Two datacenters are created:

```
Datacenter datacenter0 = createDatacenter("Datacenter_0");  
Datacenter datacenter1 = createDatacenter("Datacenter_1");
```

- ❑ Create broker:

```
DatacenterBroker broker = createBroker();
```

- ❑ Create VMs and Cloudlets and send them to broker.

```
vmList = createVM(brokerId, 5, 0);  
cloudletList = createCloudlet(brokerId, 10, 0);
```

- ❑ Start the simulation and print the results once the simulation is over.

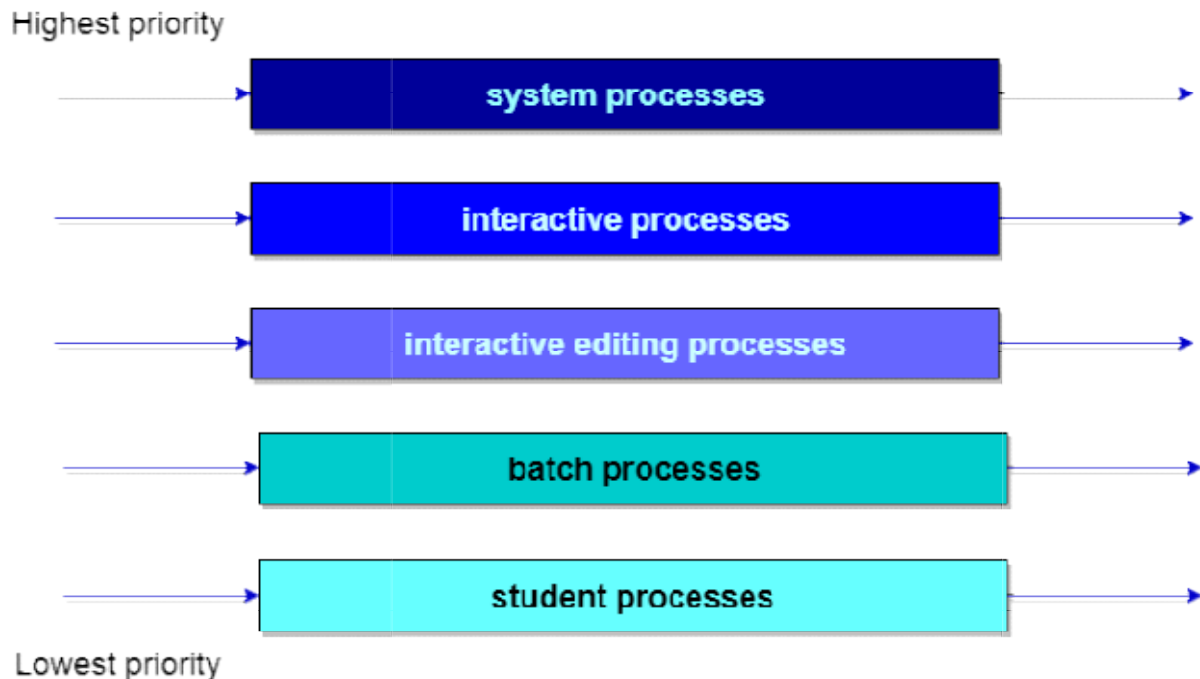


- ❑ We use a class GlobalBroker that inherits SimEntity to create the entities in runtime.
- ❑ It creates the VM list and the cloudlet list during runtime and assigns them to the broker.
- ❑ Also simulation maybe paused or resumed as needed.

4. ALGORITHM OF SCHEDULING PROCESS WITH FLOWCHART

4.1 Multi-level Priority Queue

Multi-level priority queue scheduling algorithm is used in scenarios where the processes can be classified into groups based on property like process type, CPU time, IO access, memory size, etc. One general classification of the processes is foreground processes and background processes. In a multi-level priority queue scheduling algorithm, there will be 'n' number of queues, where 'n' is the number of groups the processes are classified into. Each queue will be assigned a priority and will have its own scheduling algorithm. For the process in a queue to execute, all the queues of priority higher than it should be empty, meaning the process in those high priority queues should have completed its execution. In this scheduling algorithm, once assigned to a queue, the process will not move to any other queues.



The scheduling algorithms used at different levels of the multi priority queue are described further.

Turnaround time: It is the total time taken by process between starting and completion

Waiting time: It is the time for which process is ready to run but is not executed by CPU scheduler

Burst time: Actual execution time of the process

4.2 Proportional Shared Algorithm

Proportional Share Scheduling is a type of scheduling that pre-allocates certain amount of CPU time to each of the processes. Every job receives a fair share of the available resources, which is why it is sometimes called fair share or time shared scheduling.

Round Robin is the scheduling algorithm used by the CPU internally during execution of the process. Preemption is the added functionality to switch between the processes. A small unit of time, also known as time slice or quantum, is defined.

The ready queue works like circular queue. Each new process is added to the tail of the ready/circular queue. Time slices (any natural number) are assigned to each process in equal portions and in circular order, dealing with all process without any priority.

The main advantage of round robin algorithm is that it is starvation free i.e. every process will be executed by CPU for fixed interval of time (which is set as time slice) so in this way no process left waiting for its turn to be executed by the CPU.

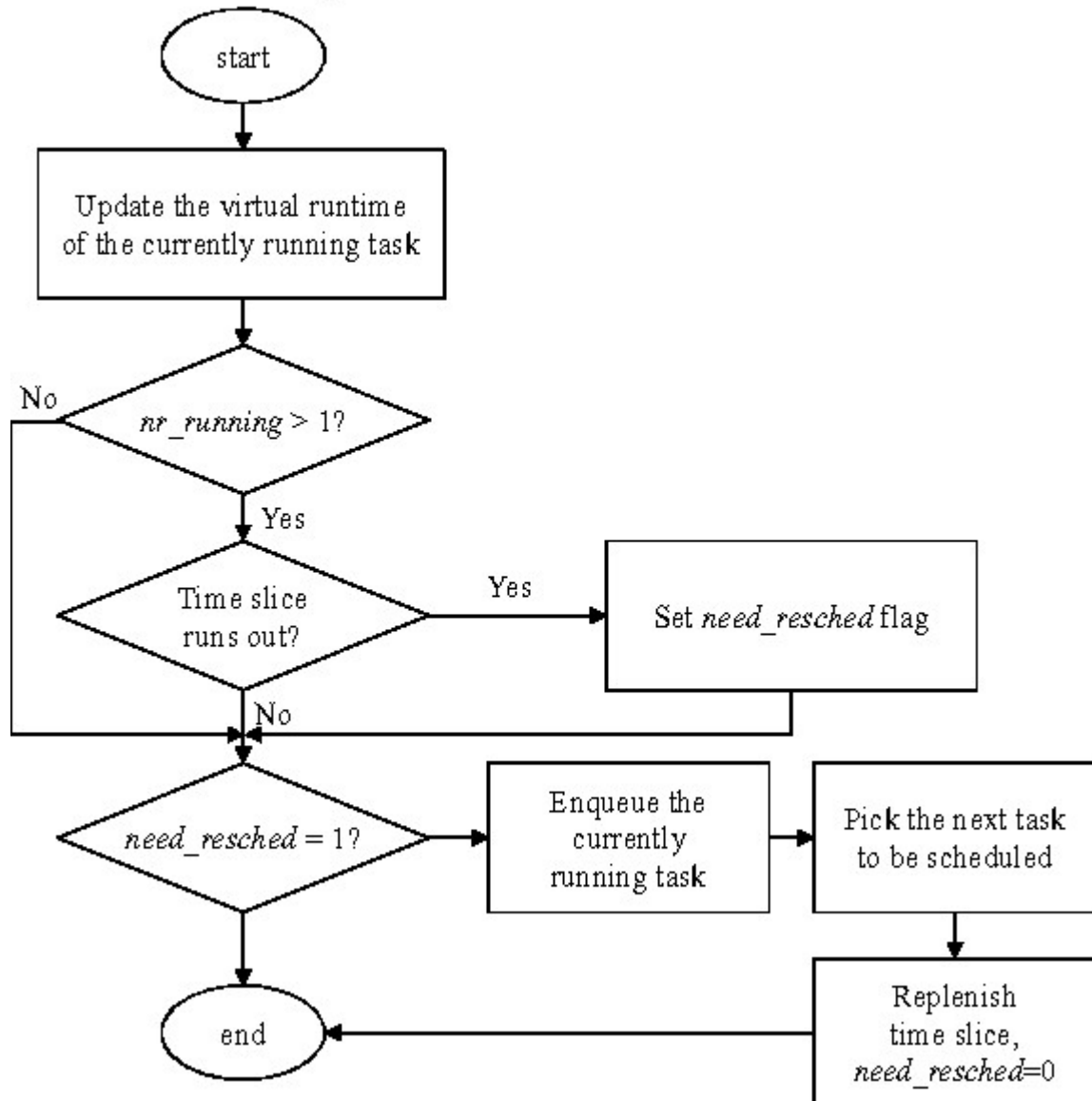
Pseudo Code:

- CPU scheduler picks the process from the circular/ready queue, set a timer to interrupt it after 1 time slice/quantum and dispatches it.
- If process has burst time less than 1 time slice/quantum,
 - Process will leave the CPU after the completion
 - CPU will proceed with the next process in the ready queue / circular queue
- Else if process has burst time longer than 1 time slice/quantum,
 - Timer will be stopped. It causes an interruption to be sent to the OS.
 - Executed process is then placed at the tail of the circular / ready queue by applying the context switch
 - CPU scheduler then proceeds by selecting the next process in the ready queue.

Major features:

- Throughput is low as a given process takes longer to complete execution due to sharing of resources.
- No starvation

For each scheduling tick



4.3 Priority Scheduling Algorithm

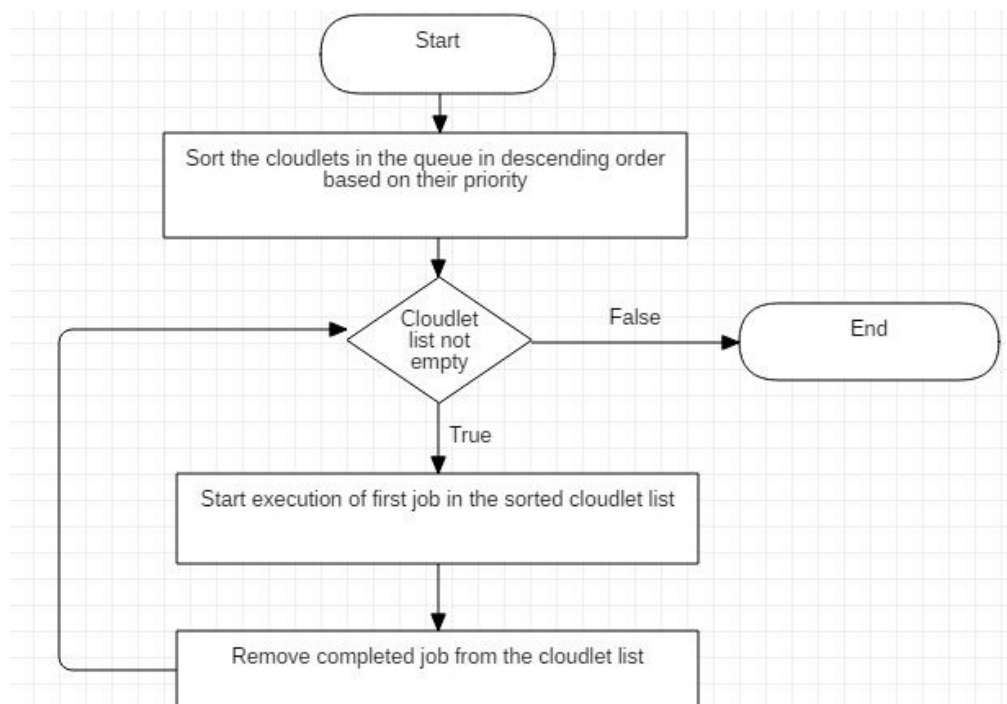
Priority scheduling used in this project is a non-preemptive algorithm. Each process is assigned a priority. Process with highest priority is to be executed first and so on. Processes with same priority are executed on first come first served basis. Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Priorities can be either dynamic or static. Static priorities are allocated during creation, whereas dynamic priorities are assigned depending on the behavior of the processes while in the system. Priorities may be defined internally or externally. Internally defined priorities make use of some measurable quantity to calculate the priority of a given process. In contrast, external priorities are defined using criteria beyond the operating system (OS), which can include the significance of the process, the type as well as the sum of resources being utilized for computer use, user preference, commerce and other factors like politics, etc.

Pseudo code:

- First input the processes with their arrival time, burst time and priority.
- Sort the processes according to arrival time, if their arrival times are same then sort them according to priority.
- Now simply apply FCFS algorithm, as described in Section 4.5.

Issues: Indefinite blocking (starvation) is a state where a process is ready to be executed, but faces a long wait in getting assigned to the CPU. It is often possible that a priority scheduling algorithm can make a low-priority process wait indefinitely. A remedy to starvation is aging, which is a technique used to gradually increase the priority of those processes that wait for long periods in the system.

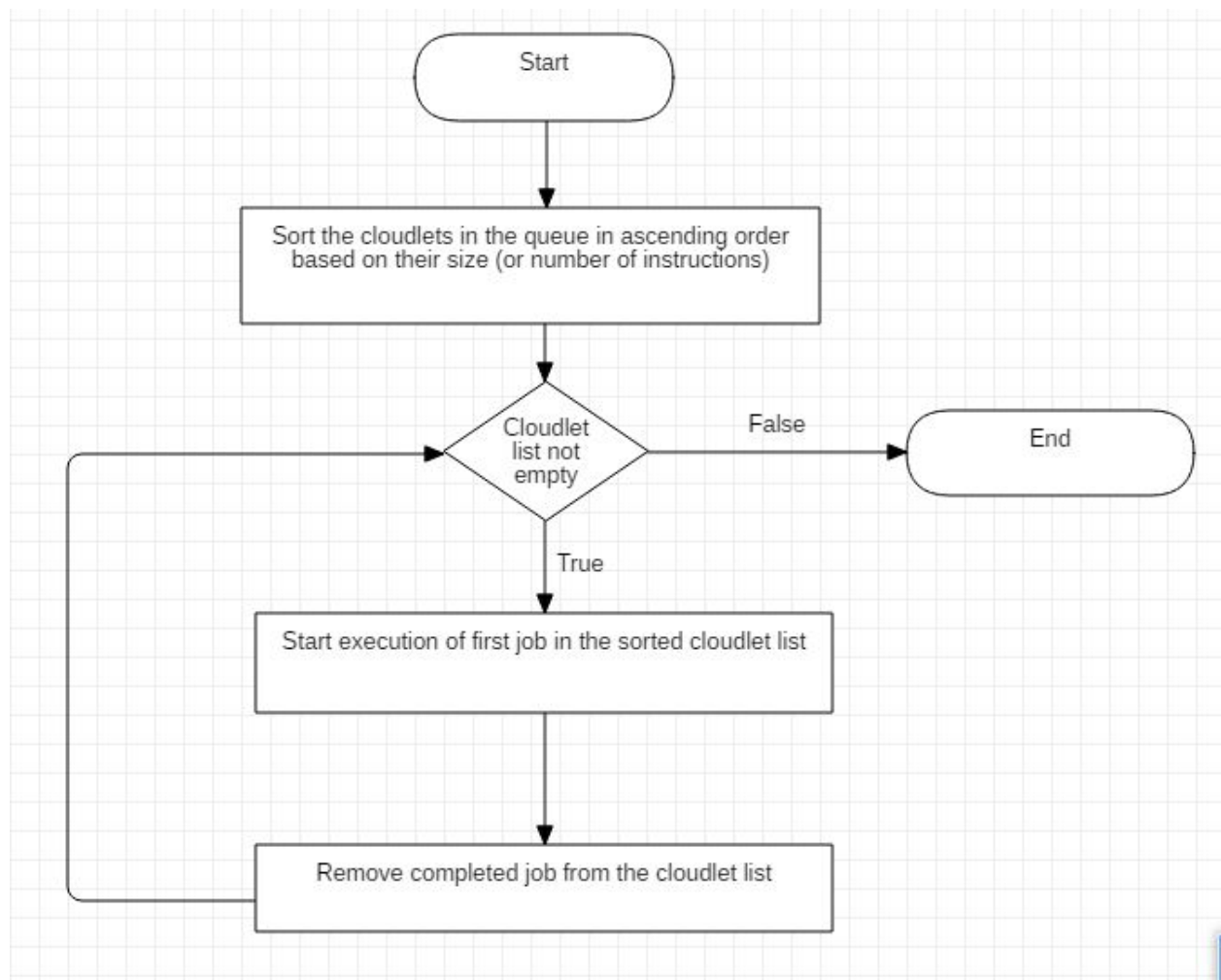


4.4 Shortest Job First Scheduling Algorithm

Shortest job first (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJN is a non-preemptive algorithm.

Features:

- Shortest Job first has the advantage of having minimum average waiting time among all scheduling algorithms.
- It is a Greedy Algorithm.
- It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of aging.
- It is practically infeasible as Operating System may not know burst time and therefore, may not sort them. While it is not possible to predict execution time, several methods can be used to estimate the execution time for a job, such as a weighted average of previous execution times. SJF can be used in specialized environments where accurate estimates of running time are available.
- This is used in Batch Systems.



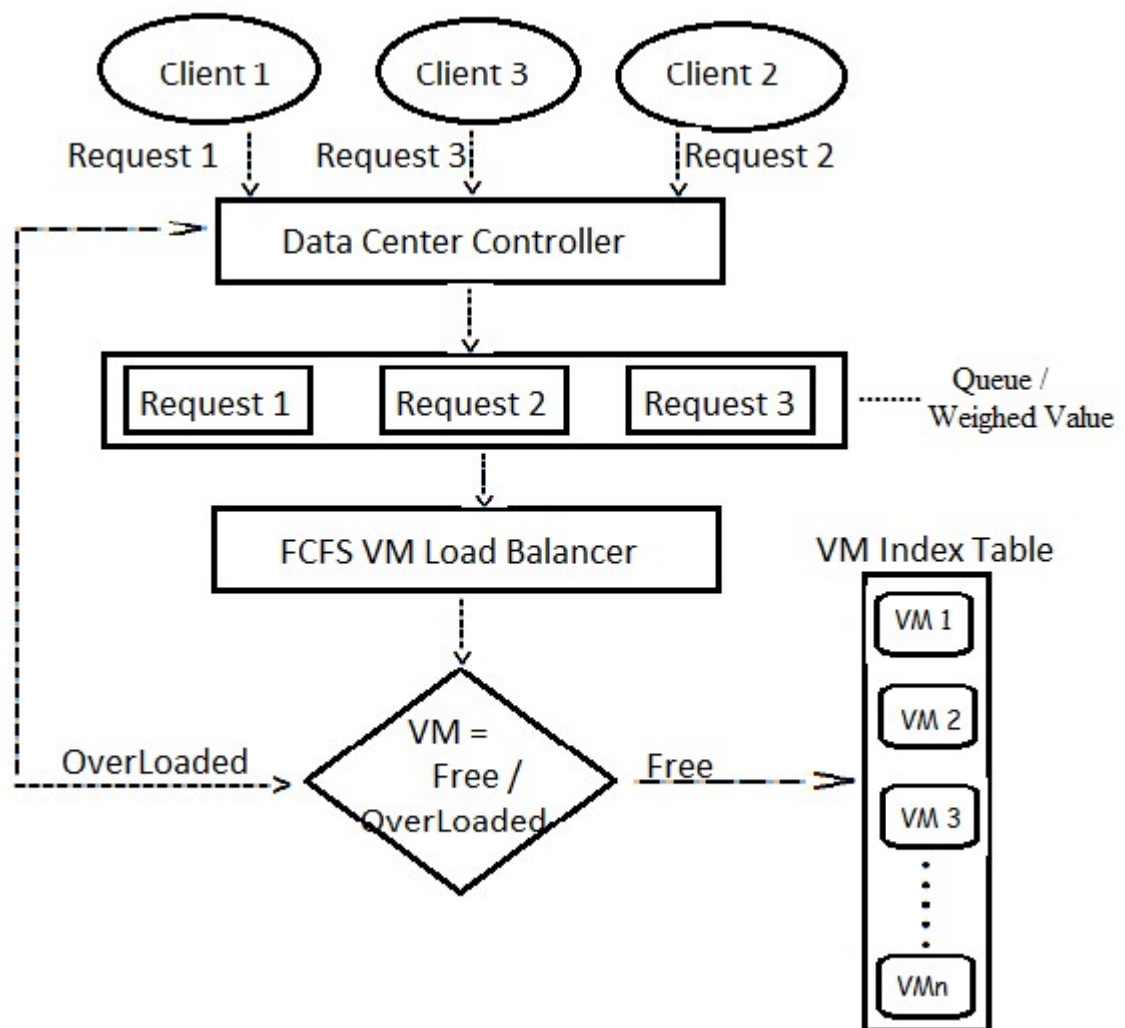
4.5 First Come First Serve Scheduling Algorithm

First-Come-First-Served algorithm is the simplest scheduling algorithm. Processes are dispatched according to their arrival time on the ready queue. Being a non-pre-emptive discipline, once a process has a CPU, it runs to completion.

The FCFS scheduling is fair in the formal sense or human sense of fairness but it is unfair in the sense that long jobs make short jobs wait and unimportant jobs make important jobs wait.

FCFS is more predictable than most of other schemes since it offers time. FCFS scheme is not useful in scheduling interactive users because it cannot guarantee good response time.

One of the major drawbacks of this scheme is that the average time is often quite long. The First-Come-First-Served algorithm is rarely used as a master scheme in modern operating systems but it is often embedded within other schemes.



5. CODE

```
import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.LinkedList;
import java.util.List;
import org.cloudbus.cloudsim.Cloudlet;
import org.cloudbus.cloudsim.CloudletSchedulerPriorityBased;
import org.cloudbus.cloudsim.CloudletSchedulerProportionalShared;
import org.cloudbus.cloudsim.CloudletSchedulerSJFSched;
import org.cloudbus.cloudsim.CloudletSchedulerSpaceShared;
import org.cloudbus.cloudsim.Datacenter;
import org.cloudbus.cloudsim.DatacenterBroker;
import org.cloudbus.cloudsim.DatacenterCharacteristics;
import org.cloudbus.cloudsim.Host;
import org.cloudbus.cloudsim.Log;
import org.cloudbus.cloudsim.Pe;
import org.cloudbus.cloudsim.Storage;
import org.cloudbus.cloudsim.UtilizationModel;
import org.cloudbus.cloudsim.UtilizationModelFull;
import org.cloudbus.cloudsim.Vm;
import org.cloudbus.cloudsim.VmAllocationPolicySimple;
import org.cloudbus.cloudsim.VmSchedulerTimeShared;
import org.cloudbus.cloudsim.core.CloudSim;
import org.cloudbus.cloudsim.provisioners.BwProvisionerSimple;
import org.cloudbus.cloudsim.provisioners.PeProvisionerSimple;
import org.cloudbus.cloudsim.provisioners.RamProvisionerSimple;
import java.util.Random;

public class MultiLevelPriorityQueue {
    private static List<Cloudlet> cloudletList;
    private static List<Vm> vmList;
    public static void main(String[] args) {
        Log.println("Starting Multilevel Priority queue...");
        Random rand = new Random();
        try {
            int num_user = 1; // number of cloud users
            Calendar calendar = Calendar.getInstance();
            boolean trace_flag = false; // mean trace events
            CloudSim.init(num_user, calendar, trace_flag);
            Datacenter datacenter0 = createDatacenter("Datacenter_0");
            DatacenterBroker broker = createBroker();
            int brokerId = broker.getId();
            vmList = new ArrayList<Vm>();
            int mips = 10000;
            long size = 10000; // image size (MB)
            int ram = 512; // vm memory (MB)
            long bw = 1000;
            int pesNumber = 1; // number of cpus
            String vmm = "Xen"; // VMM name
            Vm[] vm = new Vm[8];
            for(int i=0; i<2;i++)
            {
                mips=rand.nextInt(40000)+500;
                vm[i] = new Vm(i, brokerId, mips, pesNumber, ram, bw, size, vmm, new
CloudletSchedulerProportionalShared());
```

```

        vmList.add(vm[i]);
    }
    for(int i=2; i<4;i++)
    {
        mips=rand.nextInt(5000)+500;
        vm[i] = new Vm(i, brokerId, mips, pesNumber, ram, bw, size, vmm, new
CloudletSchedulerPriorityBased());
        vmList.add(vm[i]);
    }
    for(int i=4; i<6;i++)
    {
        mips=rand.nextInt(5000)+500;
        vm[i] = new Vm(i, brokerId, mips, pesNumber, ram, bw, size, vmm, new
CloudletSchedulerSJFSched());
        vmList.add(vm[i]);
    }
    for(int i=6; i<8;i++)
    {
        mips=rand.nextInt(5000)+500;
        vm[i] = new Vm(i, brokerId, mips, pesNumber, ram, bw, size, vmm, new
CloudletSchedulerSpaceShared());
        vmList.add(vm[i]);
    }
    broker.submitVmList(vmList);
    cloudletList = new ArrayList<Cloudlet>();
    int id = 0;
    long length = 40000;
    long fileSize = 300;
    long outputSize = 300;
    UtilizationModel utilizationModel = new UtilizationModelFull();
    Cloudlet[] cloudlet=new Cloudlet[80];
    //Proportional Share Scheduling
    for(int k=0;k<20;k++)
    {
        cloudlet[k] = new Cloudlet(k, length, pesNumber, fileSize, outputSize,
utilizationModel, utilizationModel, utilizationModel);
        cloudlet[k].setUserId(brokerId);
    }
    int vid=0;
    for(int i=0;i<20;i++)
    {
        if( vid==2)
        {
            vid=0;
        }
        cloudlet[i].setVmId(vid);
        cloudletList.add(cloudlet[i]);
        vid++;
    }
    //Priority implementation
    for(int k=20;k<40;k++)
    {
        cloudlet[k] = new Cloudlet(k, length, pesNumber, fileSize, outputSize,
utilizationModel, utilizationModel, utilizationModel, (k+1)%3+1);
        cloudlet[k].setUserId(brokerId);
    }

```

```

vid=2;
for(int i=20;i<40;i++)
{
    if( vid==4)
    {
        vid=2;
    }

    cloudlet[i].setVmId(vid);
    cloudletList.add(cloudlet[i]);
    vid++;
}
//SJF implementation
id = 0;
length = 40000;
fileSize = 300;
outputSize = 300;
for(int k=40;k<60;k++)
{
    length=Math.abs(rand.nextInt())%50000;
    cloudlet[k] = new Cloudlet(k, length, pesNumber, fileSize, outputSize,
utilizationModel, utilizationModel, utilizationModel);
    cloudlet[k].setUserId(brokerId);
}
vid=4;
for(int i=40;i<60;i++)
{
    if( vid==6)
    {
        vid=4;
    }
    cloudlet[i].setVmId(vid);
    cloudletList.add(cloudlet[i]);
    vid++;
}
//FCFS implementation
for(int k=60;k<80;k++)
{
    cloudlet[k] = new Cloudlet(k, length, pesNumber, fileSize, outputSize,
utilizationModel, utilizationModel, utilizationModel);
    cloudlet[k].setUserId(brokerId);
}
vid=6;
for(int i=60;i<80;i++)
{
    if( vid==8)
    {
        vid=6;
    }
    cloudlet[i].setVmId(vid);
    cloudletList.add(cloudlet[i]);
    vid++;
}
broker.submitCloudletList(cloudletList);
CloudSim.startSimulation();
CloudSim.stopSimulation();
List<Cloudlet> newList = broker.getCloudletReceivedList();

```



```

        printCloudletList(newList);
        Log.println("\nExecution finished!");
    } catch (Exception e) {
        e.printStackTrace();
        Log.println("Unwanted errors happen");
    }
}

private static Datacenter createDatacenter(String name) {
    List<Host> hostList = new ArrayList<Host>();
    List<Pe> peList = new ArrayList<Pe>();
    int mips = 3000000;
    peList.add(new Pe(0, new PeProvisionerSimple(mips))); // need to store Pe id and MIPS Rating
    int hostId = 0;
    int ram = 8048; // host memory (MB)
    long storage = 1000000; // host storage
    int bw = 10000;
    hostList.add(
        new Host(
            hostId,
            new RamProvisionerSimple(ram),
            new BwProvisionerSimple(bw),
            storage,
            peList,
            new VmSchedulerTimeShared(peList)
        ); // This is our machine
    String arch = "x86"; // system architecture
    String os = "Linux"; // operating system
    String vmm = "Xen";
    double time_zone = 10.0; // time zone this resource located
    double cost = 3.0; // the cost of using processing in this resource
    double costPerMem = 0.05; // the cost of using memory in this resource
    double costPerStorage = 0.001; // the cost of using storage in this resource
    double costPerBw = 0.0; // the cost of using bw in this resource
    LinkedList<Storage> storageList = new LinkedList<Storage>(); // we are not adding SAN
    DatacenterCharacteristics characteristics = new DatacenterCharacteristics(
        arch, os, vmm, hostList, time_zone, cost, costPerMem,
        costPerStorage, costPerBw);
    Datacenter datacenter = null;
    try {
        datacenter = new Datacenter(name, characteristics, new
VmAllocationPolicySimple(hostList, storageList, 0);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return datacenter;
}

private static DatacenterBroker createBroker() {
    DatacenterBroker broker = null;
    try {
        broker = new DatacenterBroker("Broker");
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
    return broker;
}

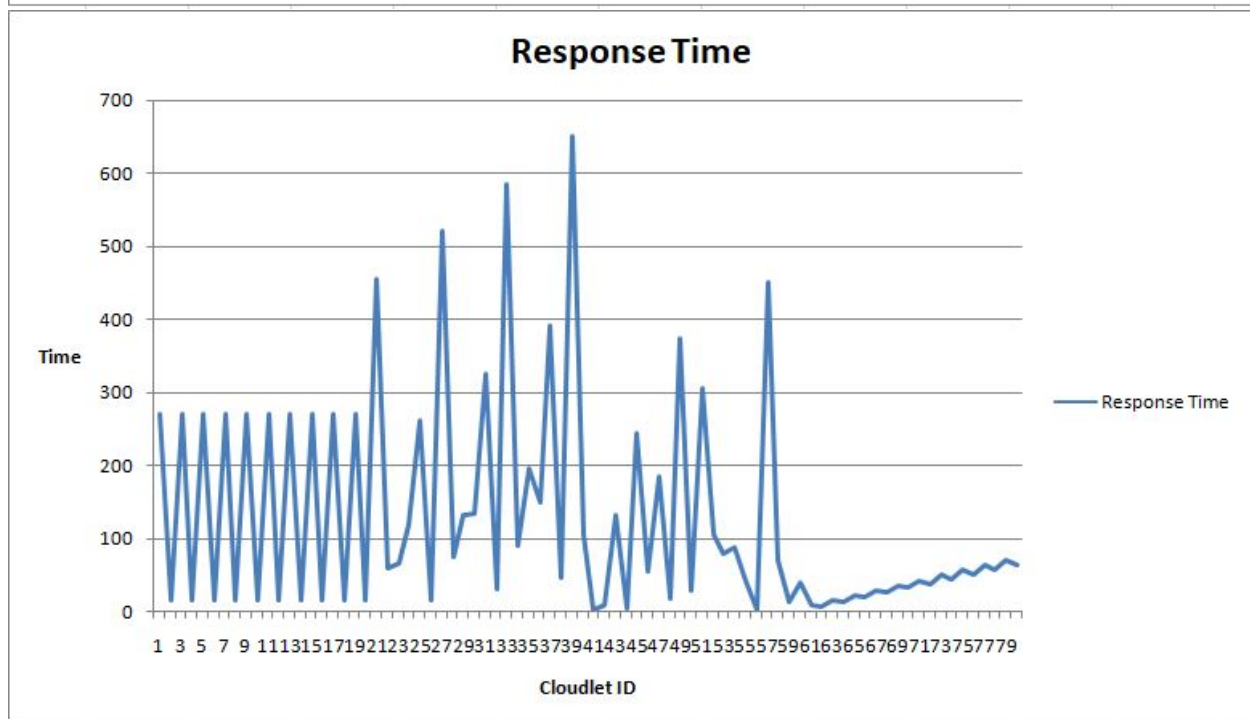
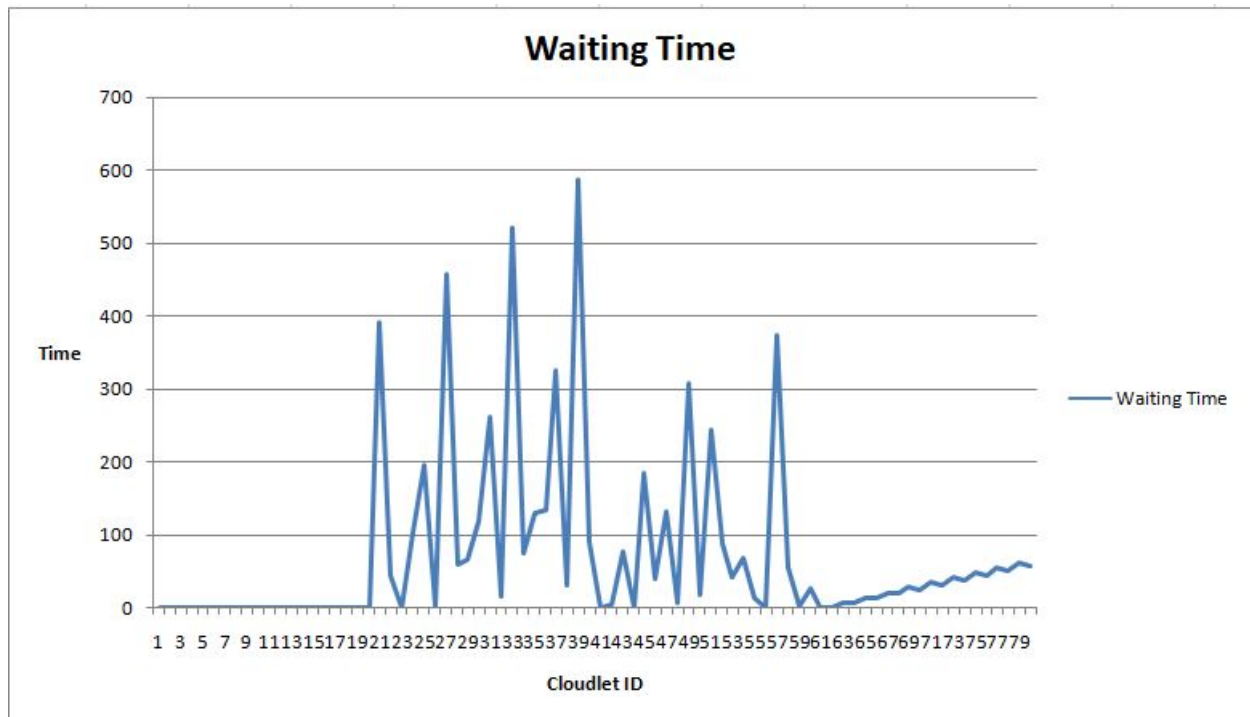
```

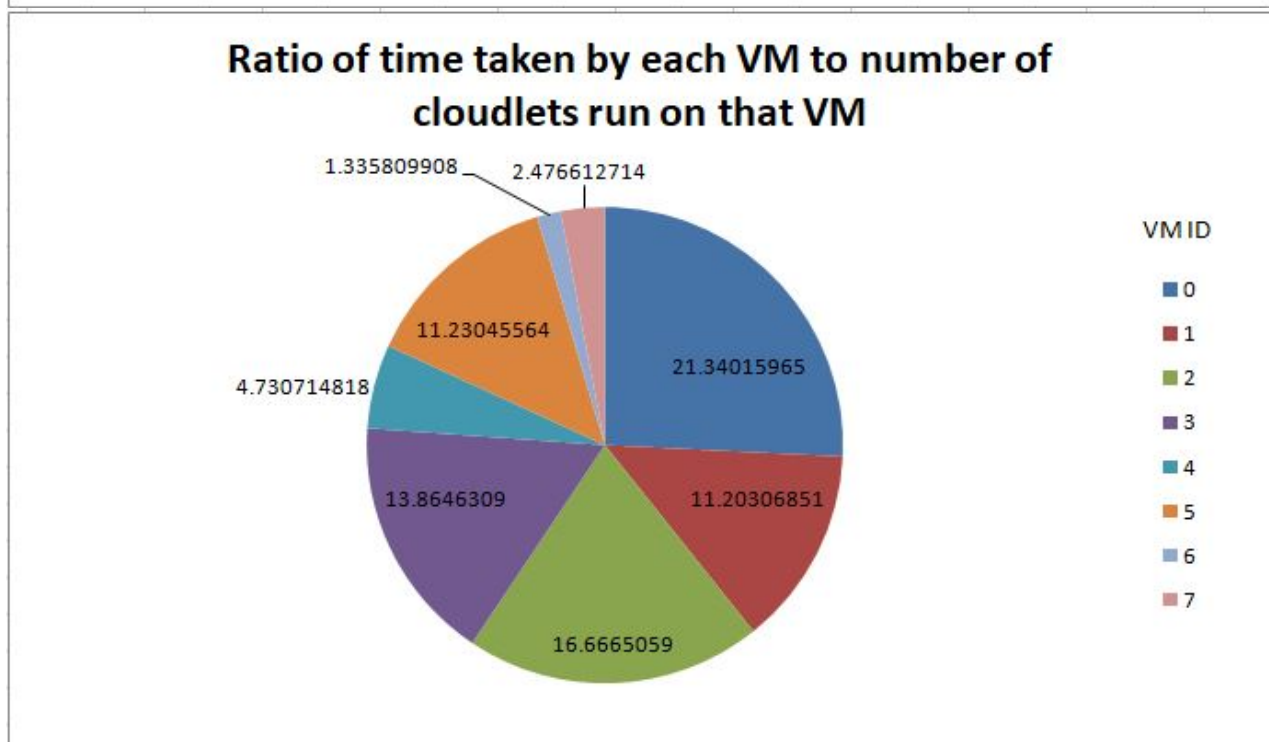
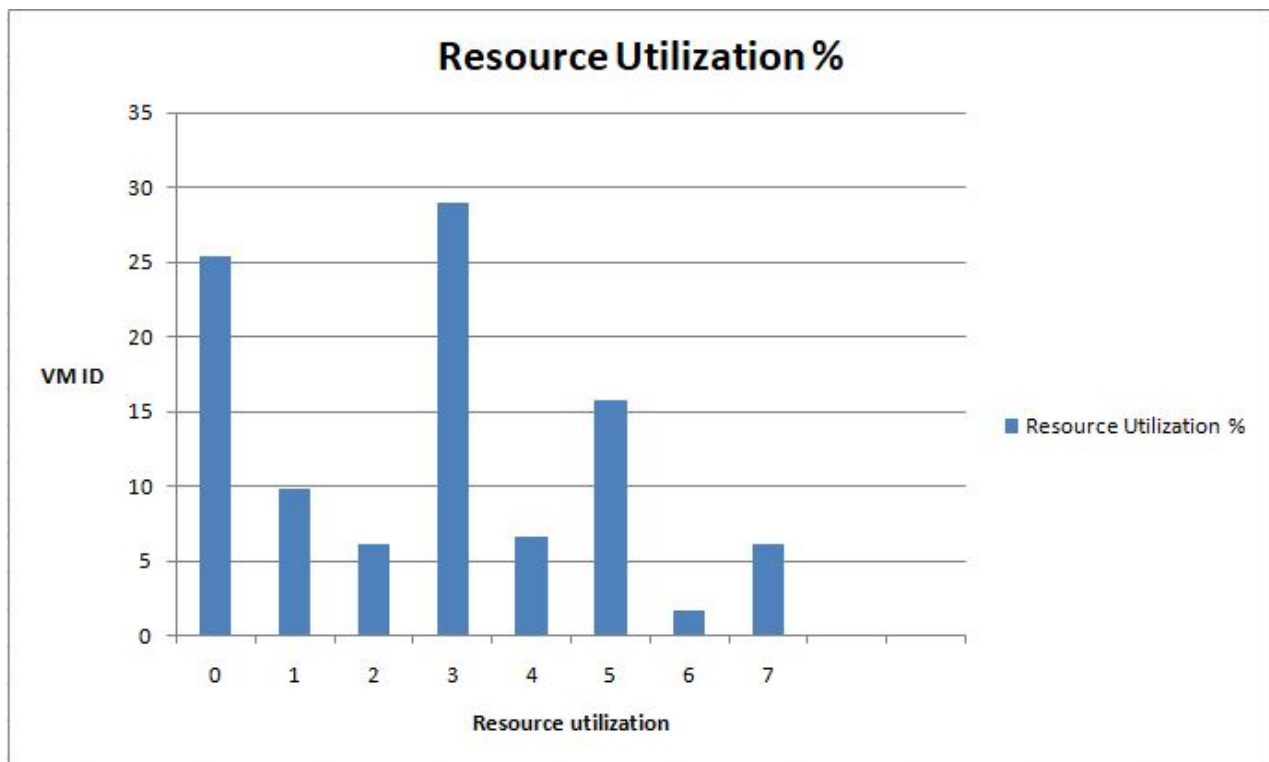
```

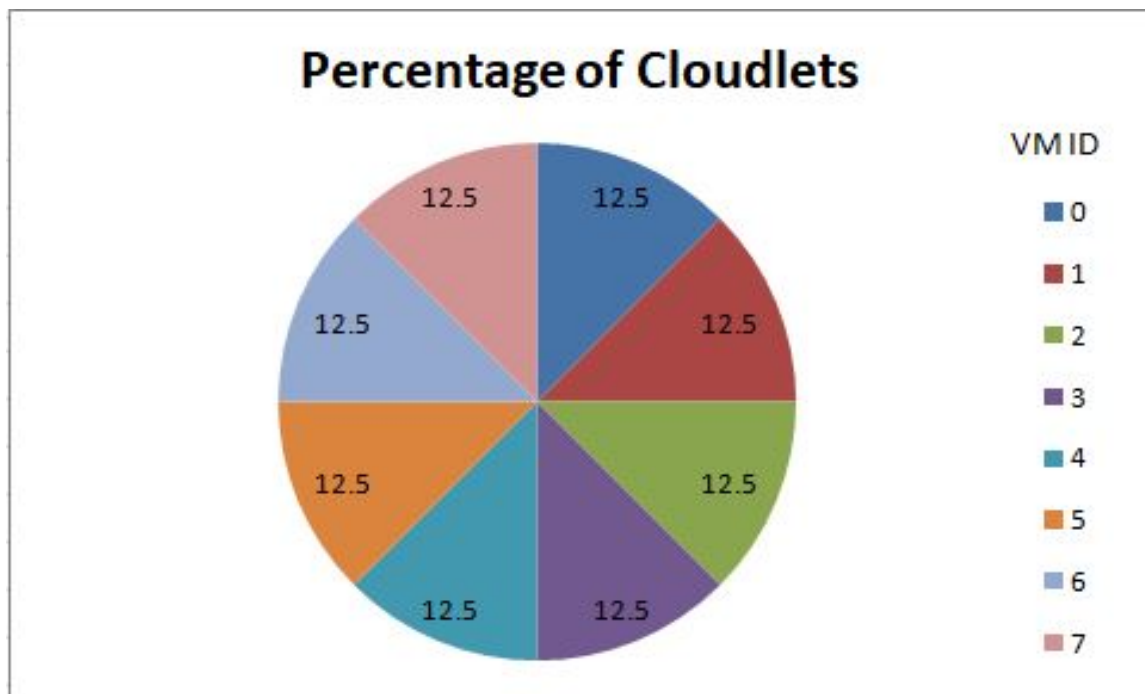
private static void printCloudletList(List<Cloudlet> list) {
    int size = list.size();
    Cloudlet cloudlet;
    String indent = "\t";
    Log.println();
    int index = 0;
    Log.println("===== OUTPUT =====");
    Log.println("Cloudlet ID" + indent + indent + "STATUS" + indent
        + indent + "Data center ID" + indent + indent + "VM ID" + indent + indent +
"Time" + indent
        + indent + "Start Time" + indent + "Finish Time" + indent + "Length" + indent +
"Priority");
    DecimalFormat dft = new DecimalFormat("###.##");
    index = 0;
    for (int i = 0;; i++) {
        cloudlet = list.get(i%80);
        if(index==cloudlet.getCloudletId()){
            Log.print(indent + cloudlet.getCloudletId() + indent + indent);
            if (cloudlet.getCloudletStatus() == Cloudlet.SUCCESS) {
                Log.print("SUCCESS");
                Log.println(indent + indent + cloudlet.getResourceId()
                    + indent + indent + indent + cloudlet.getVmId()
                    + indent + indent
                    + dft.format(cloudlet.getActualCPUTime()) + indent
                    + indent + dft.format(cloudlet.getExecStartTime())
                    + indent + indent
                    + dft.format(cloudlet.getFinishTime())
                    + indent + indent
                    + cloudlet.getCloudletLength()
                    + indent + indent
                    + (cloudlet.getPriority()!=-1?cloudlet.getPriority():"-"));
            }
            index++;
            if(index==size){
                return;
            }
        }
    }
}
}
}

```

6. GRAPHS







7. CONCLUSION

The assignment given to us has been successfully completed. During the course of completion, we were exposed to the different functionalities of CloudSim simulator. Multilevel priority queue consists of the implementation of four algorithms: Fair Sharing, Priority based, Shortest Job First and FCFS. The cloudlets are executed in the Virtual machines based on the scheduling algorithm assigned to them.

8. REFERENCES

8.1. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms “,Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov1, C’esar A. F. De Rose and RajkumarBuyya, SOFTWARE - PRACTICE AND EXPERIENCE Softw. Pract. Exper. 2011; 41:23–50, DOI: 10.1002/spe.995, 24 August 2010