Design & Experimental Evaluation of Actor based Cloud Simulator

Masters Project Report

Semester of Graduation: Spring 2019

Project Guide: Prof. Mark Grechanik

Second Evaluator: Prof. Ugo Buy

Department of Computer Science

University of Illinois at Chicago

Author	**Adarsh Ratnakar Hegde**
UIN	**651125237**
Date	**4/30/2019**
Version	**1.0**

TABLE OF CONTENTS

1. INTRODUCTION

The goal of the project is to develop a simulation tool for cloud environments. The user of the simulator would be able to simulate conditions in the cloud infrastructure to understand the behavior of the system using the approximation of the actual operations.

With the rapid adoption of cloud computing it has become imperative to evaluate the performance of cloud infrastructures to ensure the quality of service and to predict the behavior. The cloud provider makes various business decisions like marking the cost, deciding the availability of services and building the architecture of the system. The end-user too chooses between various providers based on the suitability of the services to their demands. It is difficult to make these decisions without actual execution of applications. Using real cloud resources to test can be cost and time intensive, especially while testing the infrastructure to the limits. Therefore, the use of simulation provides an opportunity to mitigate the costs and test the system under flexible conditions.

In this project (CloudSimulatorAkka) we have built a cloud simulation tool that can be used to flexibly build cloud infrastructure, extended further to incorporate finer complexities within cloud computing and run different types of workloads to obtain cost and duration of operation. The simulator implements various entities representing real-world cloud infrastructure resources (datacenter, host, virtual machine, scheduling policies, etc.). Furthermore, it defines the communication between them. The user can organize these entities in different configurations and execute workloads representing applications. Also, the simulator captures various metrics like cost, execution time, etc. Finally, we have performed several experiments using different sizes of infrastructures and workloads and compared it to other cloud simulation tools.

The biggest distinction from other cloud simulators is the use of the Akka system to build the simulator upon. Akka is a toolkit for building distributed, concurrent and message-driven

applications. One can imagine a cloud computing system as a collection of distributed entities with defined interactions. Akka's actor model perfectly mirrors this with actors communicating with each other using message passing.

2. CLOUD SIMULATOR

2.1. Simulators

Simulators are widely used to imitate the operation of complex systems. They provide clear insights into the functioning of the system in a risk-free and cost-effective manner. These insights could include behavior in the system under various input models and/or results of testing. Some applications for simulator systems are in the areas of traffic simulation, airline reservation simulation and sports-training simulation. For example, simulation is heavily used in rail transportation to model the rail traffic. The results of these experiments drive the decisions behind the allocation of lanes, routing and capacity of the network.

A cloud simulator can thus help stakeholders analyze how the cloud infrastructure and the applications deployed on this infrastructure will perform under varying conditions. This helps all parties to make informed decisions.

2.2. Cloud Sim – A Java based Cloud Simulator

Cloud Sim is developed in the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, at the Computer Science and Software Engineering Department of the University of Melbourne [1]. The initial release was built on the Sim Java's event simulation engine and follows its threading model for the creation of the Simulation Entities. Every entity of Cloud Sim which extends the Sim_Entity of Sim Java is capable of sending and receiving the messages through the Sim Java's shared event message queue. Each Sim_Entity component represents a thread. Multi-threaded nature of SimJava leads to performance overhead with the increase in system size. Performance degradation is caused by the excessive context switching between the threads[2].

2.3. Use of Scala and Akka Framework

Scala [1] [2] is a multi-paradigm programming language supporting use of imperative, object-oriented and functional styles. It was designed to be fully interoperable with Java and the code written in Scala is compiled into Java bytecode.

Scala was chosen for the Cloud Simulator because it allowed development of a modular system. The components in the system can be easily extended with the use of traits. The user can add finer details of the cloud computing system with ease. The existence of a well-developed standard library and the interoperability with Java libraries aids quick development and easy adoption for the end user. Also, the support for functional programming makes the code concise and easier to perform concurrent operations.

The Akka Framework [3] defines the concept of actors. An actor is defined as a container for State, Behavior, a Mailbox, Child Actors and a Supervisor Strategy. Akka executes actors using threads where each actor is accessed in a single threaded manner only. This ensures the internal state of the actor is always consistent. Also, it allows actors to store mutable state which reduces the overhead of creating instances for every state change operation.

The framework provides concurrency through message passing between actor instances. Message passing behaves differently from function calls. Unlike functions, messages don't have return values. An actor after sending a message to another actor doesn't wait for a return and continues its execution in a non-blocking fashion. Also, in event of a message, the thread of execution is not passed to the receiving actor. Thus, the receiving actor's state is not accessed by multiple threads.

In Akka, message passing is established using the concept of a sender and receiver. Every actor has a mailbox; the sender actor delivers the message to the mailbox of the receiver actor. The receiver processes the messages in a First-In-First-Out (FIFO) fashion, as a result allowing only

single-threaded, concurrent operations on its state, thus negating the disadvantages of mutability. The framework ensures the single-threaded access to the internal state, thereby freeing the client code of the responsibility to ensure concurrent access.

The conceptual similarity in the operation of Akka actors and distributed objects make Akka a great fit for building the Cloud Simulator. Cloud entities like Datacenter, Host, Switch, etc. can be implemented as actors. Furthermore, the interactions between these entities in the form of network packets can be established using Akka messages. The relationships between the entities can be preserved too, for example, the host actor can be a child of the datacenter actor to show that resources like hosts, virtual machines, scheduling policies etc. are resources encapsulated by a datacenter in the real-world cloud system.

3. DESIGN

We present the design of the system in Figure 1 using the following key components [4] in the simulator.

The Network topology is the topological structure of cloud resources in the system. If viewed as a graph, the topology includes the entities like hosts, switches, datacenters etc. as the nodes and the communication channels between them as the edges.

Message delivery is the process of transfer of messages from one entity in the system to another. This is established using message passing between Akka actors.

Cloud Information Services (CIS) is the registry for datacenters in the system. All datacenters upon instantiation register themselves with the CIS. Any entity in the system can ask the CIS to send the list of active datacenters in the system.

A datacenter is a physical or logical encapsulation of cloud resources. Using the idea of a datacenter resources like hosts, virtual machines etc. can grouped together under the same datacenter.

Load Balancer is responsible to distribute the incoming cloud workload across the datacenters. It achieves this using a cloud policy called Datacenter Selection Policy.

A host is the hardware machine that provides the computing resources for a virtual machine to run.

A virtual machine is an emulation of a computer. It uses the resources of the host upon which it is allocated for execution. At any moment the resources used by virtual machines can't be more than the resources available at the host.

A switch is a networking device that connects devices on a computer network by using packet switching to receive, process, and forward data to the destination device. The simulator defines switches to perform packet switching between simulation entities.

A workload is the amount of work the user wants to run on the cloud system.

The user sends the workload request via the network to the cloud system. The payload of the request contains the details of the type of work that is requested. We define two types of payloads - VM payload and Cloudlet payload. The VM payload defines the request for creation of a virtual machine. The Cloudlet payload defines the request for deployment of an application on virtual machines.

The cloud policies are the algorithms that are implemented by the system to make key decisions. The user of the simulator can inject their implementation into the system as dependencies to actor entities.

VM allocation is the process of creating virtual machines on host instances (within a datacenter) that satisfy the computing resource requirements and configurations of the VM payload.

VM scheduling is the strategy using which virtual machines are scheduled for execution based on resources like time and processing elements.

Datacenter selection is the process using which the load balancer chooses the datacenter to send the workload to.

Cloudlet scheduling is the strategy using which cloudlets are scheduled for execution based on resources like time and processing elements.

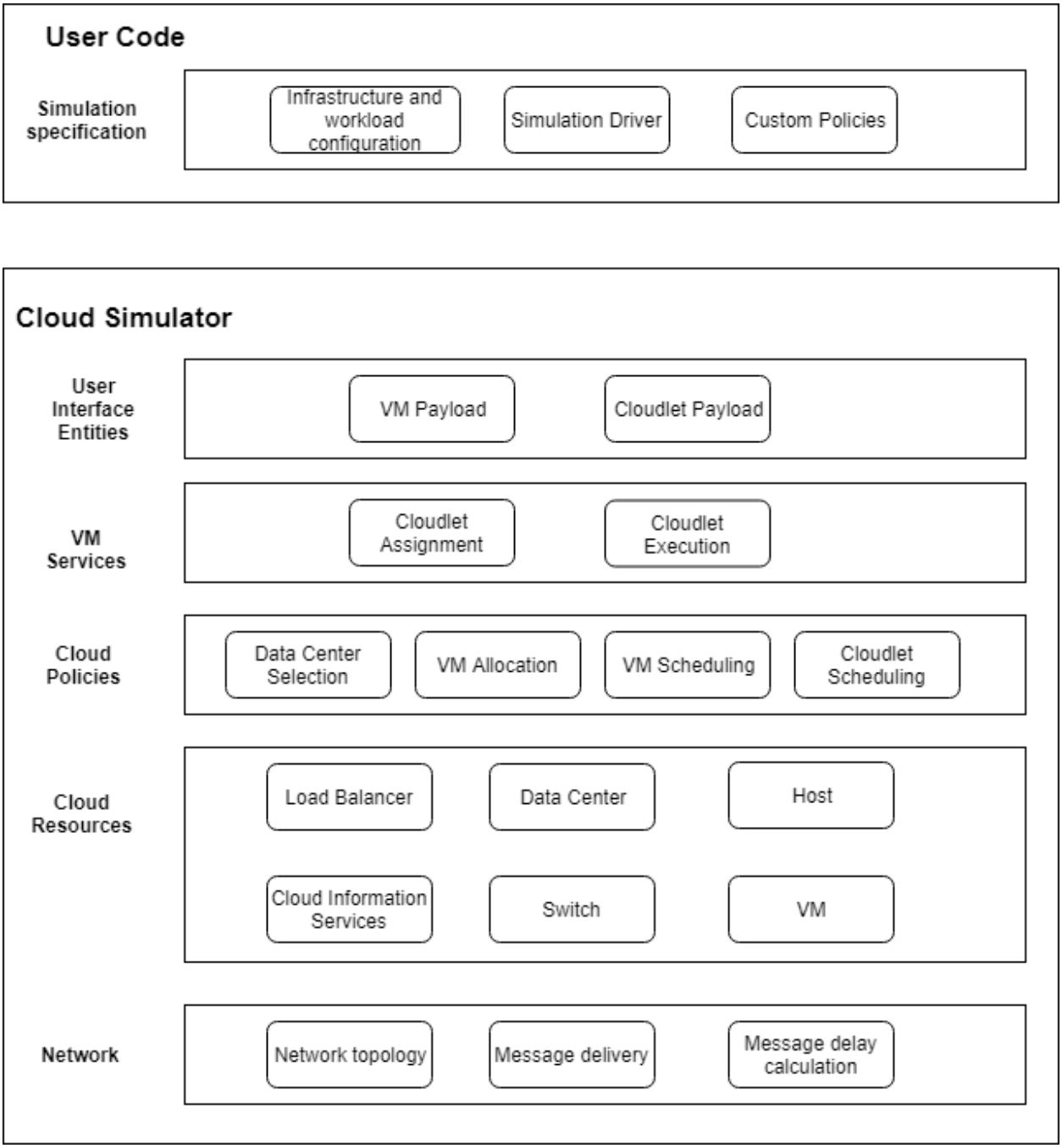

Figure 1: Layered Cloud Simulator Architecture.

Figure 1 shows the layered design of the Cloud Simulator framework along with its architectural components. The Cloud Simulator layer forms the bulk of the framework, providing the capability

to model real-world cloud infrastructure entities and policies. It also provides the User interfacing entities used by the User Code layer to exchange data with the Simulator layer. The User Code layer provides the capability to the end-user to setup the cloud infrastructure and provide custom policies.

The configuration is used to detail the architecture of the cloud system. It outlines the cloud resources (datacenters, hosts, switches, etc.), VM workloads and cloudlet workloads that will be created in the system.

The Simulation driver is an Akka actor that drives the execution of the Cloud Simulator. Here the user creates the actors for entities within the system and injects policies (Datacenter Selection, VM Allocation, VM Scheduling, Cloudlet Scheduling, etc.) as dependencies. Implementations of the policies are provided within the framework and are easily extendable.

The data exchange with the cloud setup is established using message passing between the Simulation actor and various entity actors. The workload data is passed using two user interface entities – VM Payload and Cloudlet Payload. Both specify the details, resource requirements and policy requirements for virtual machines and cloudlets respectively. The VM Payload is used to create VM actors in the system and Cloudlets are assigned to these VM actors.

A cloudlet represents any type of application workload that the users of the cloud system want to execute in the cloud computing system. Just like a cloud customer deploys applications on virtual machines, a cloudlet is assigned to a VM actor for execution.

The Cloud Simulator organizes all the actors in a similar hierarchy as a real-world cloud infrastructure. Just like a datacenter encapsulates computing resources in a cloud infrastructure, the datacenter actor acts as the parent for actors representing computing resources (hosts, switches and policies) within a datacenter. A similar relationship exists between a host and the entities (VM, VM Allocation Policy, etc.) contained within it. This relationship between the actors

makes it easy to organize the actors in the system and establish communication between them. The parent actor is responsible for supervising child actors and thus, within the Simulator, all parent actors are tasked with creating, managing and terminating respective child actors.

Further in this section we look at the design of various entities that are a part of the cloud system.

3.1. Cloud Policies

3.1.1. Datacenter Selection

The Cloud Information Services (CIS) actor is created at the beginning of the simulation and is the entity that maintains information about all Datacenters in the system. A Datacenter actor upon creation registers itself with the CIS. This way the system can keep a track of all the Datacenters in the system.

As discussed earlier, the user of the cloud simulator sends two types of workloads - VM and cloudlet. The workload defines the resource requirements and policies required by the virtual machine or cloudlet. For instance, the VM payload request is sent by the user to the simulator. The request is received by the load balancer. When a load balancer (LB) receives a VM request to allocate virtual machines, it needs to keep forwarding the request to datacenters in the system until all the virtual machines have been allocated on hosts. To begin with, the LB requests the CIS for the list of datacenters. The CIS responds to the request by sending the list of all the datacenters that have registered with it. The LB uses the Datacenter Selection policy to choose the datacenter to which the request should be forwarded to. Depending on whether the selected Datacenter can allocate all the VMs or not, the Datacenter Selection policy can be consulted again to select a different Datacenter for the remaining VM payloads.

The Datacenter Selection policy is implemented for the entire cloud simulation system. It has a dependency that implements the actual selection algorithm. This dependency is injected into the actor during creation. Thus, the end-user of the Simulator has the flexibility to implement the policy as their choice. A Simple Datacenter Selection policy implementation is provided within the framework.

The same procedure is used while identifying datacenters for deploying cloudlets.

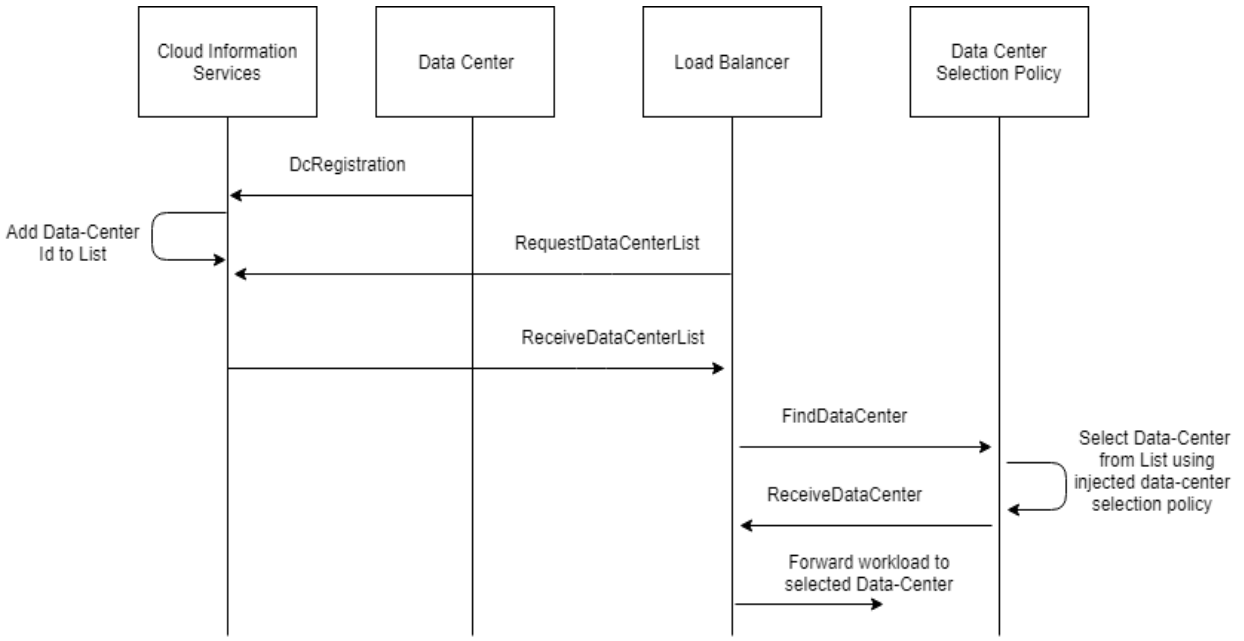

Figure 2: Datacenter Selection Policy operation.

3.1.2. VM Allocation

In Figure 3 we see that the LB sends the VM Allocation Request to the selected datacenter through the root switch. The datacenter partially/completely allocates the virtual machines within its hosts and returns the list of VM payloads for which the allocation failed. The LB then attempts to allocate these VMs at other datacenters.

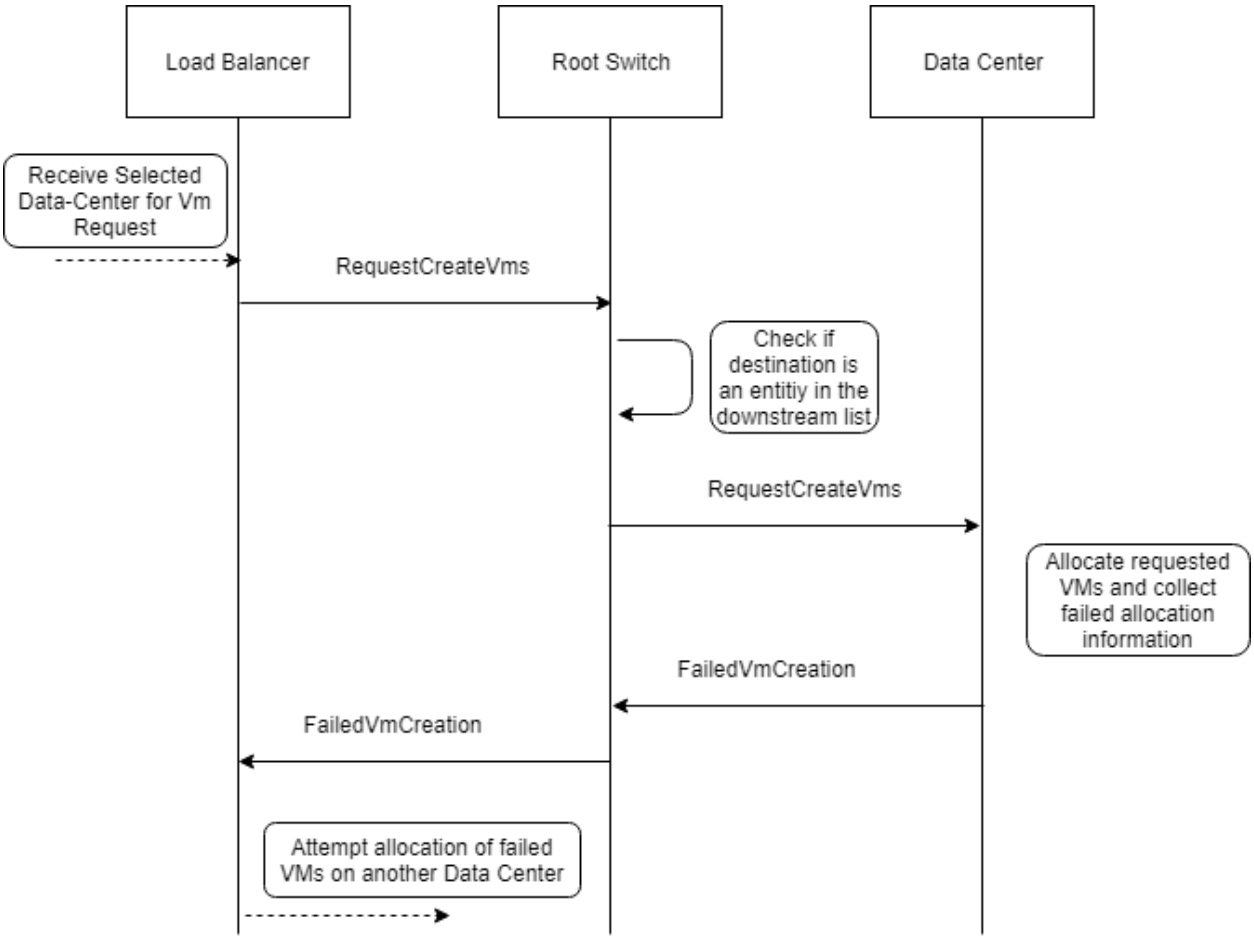

Figure 3: VM Allocation message flow from Load Balancer to selected Datacenter.

The VM Allocation Policy is defined at the datacenter level. Thus, each datacenter can allocate virtual machines uniquely. The datacenter forwards the list of VM payloads to the VM Allocation Policy actor. The VM allocation policy actor requests each host to send its resource status. The response from the host includes the processing elements (cores), RAM, storage and bandwidth available. The VM allocation policy actor has a dependency which implements the algorithm used to decide the allocation of VMs on hosts. A simple implementation of this policy is provided in the framework. It assigns the VM payloads to the hosts in a First-Come-First-Serve basis. The assignment is made only when the host satisfies all the requirements of the VM payload. Another example of an allocation strategy would be in a Map/Reduce application where the performance of the application is dependent on the strategy of allocation of virtual machines to hosts. The

preferred strategy is to allocate virtual machines logically closer to each other either in the same host or within the same rack. To incorporate these requirements, the allocation policy can be extended by the user to implement more sophisticated methods. Finally, the allocation results (including successful allocation and failed allocation) are returned to the datacenter. Figure 4 depicts the above process.

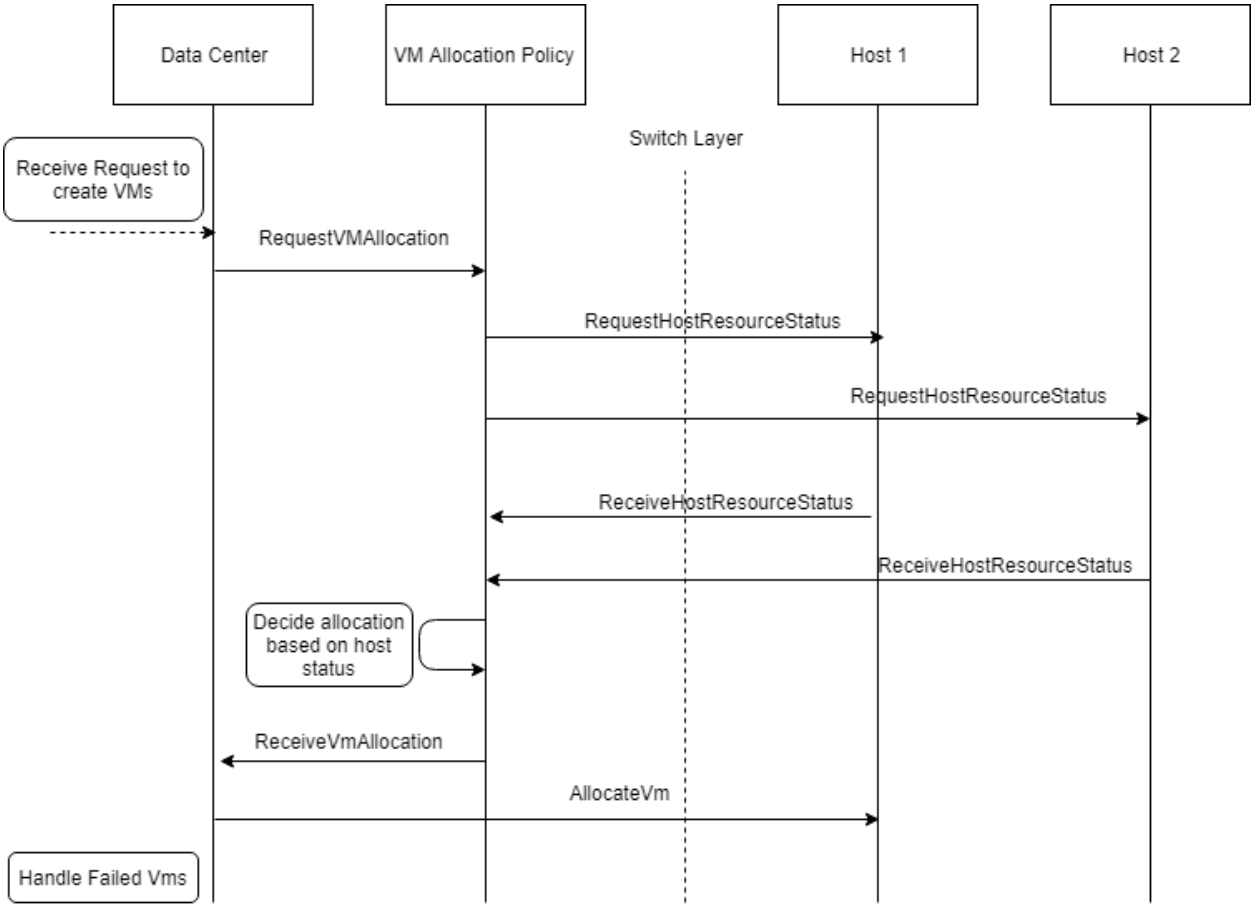

Figure 4: Allocation of VMs on hosts.

The datacenter receives the VM Allocation result from the VM Allocation Policy actor and sends the list of VMs to the LB that failed the allocation process. The LB creates a new request for the these VMs and restarts the allocation process.

Figure 5 shows how the entire message passing is conducted by the LB and handling of failed allocation.

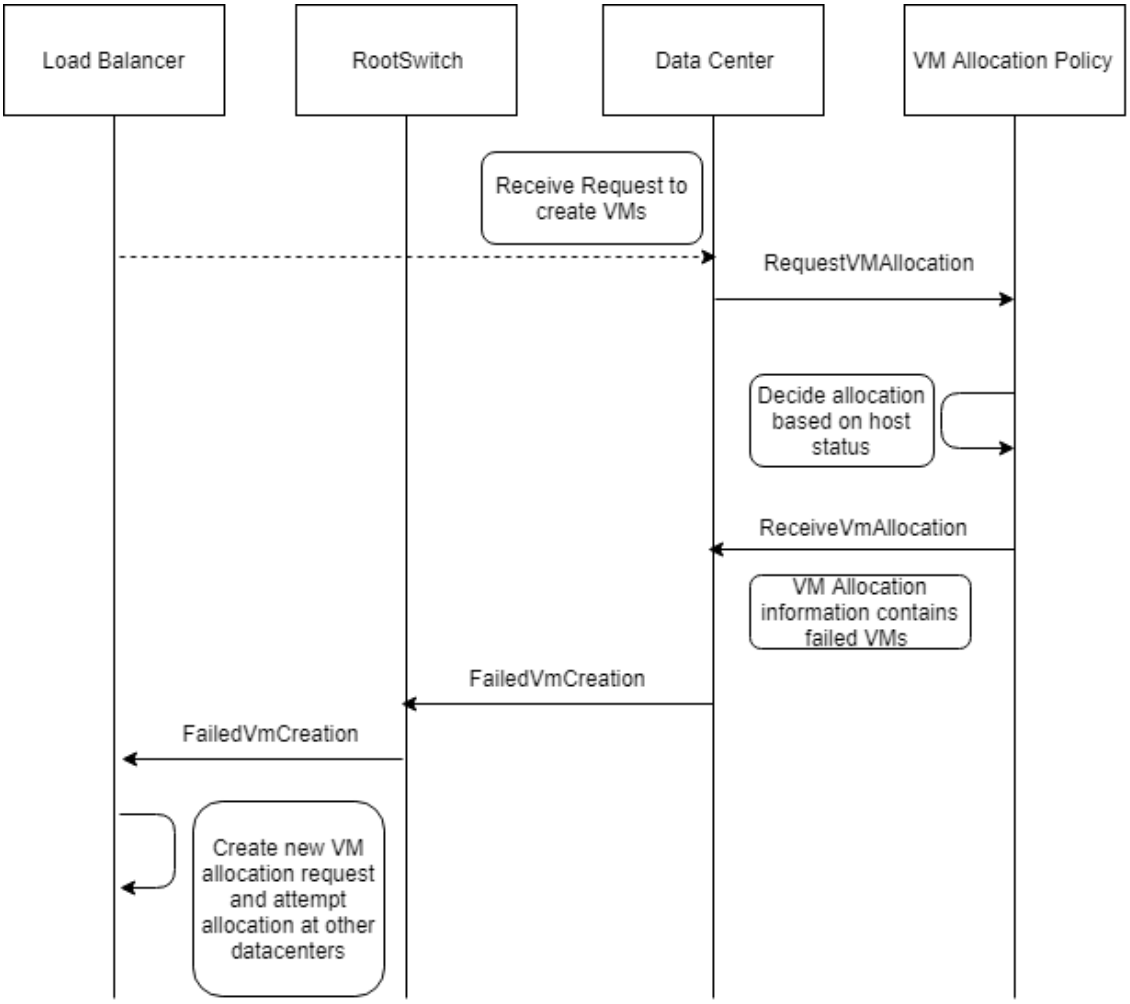

Figure 5: Failed allocation of VMs are handled by Load Balancer.

3.1.3. Cloudlet Assignment

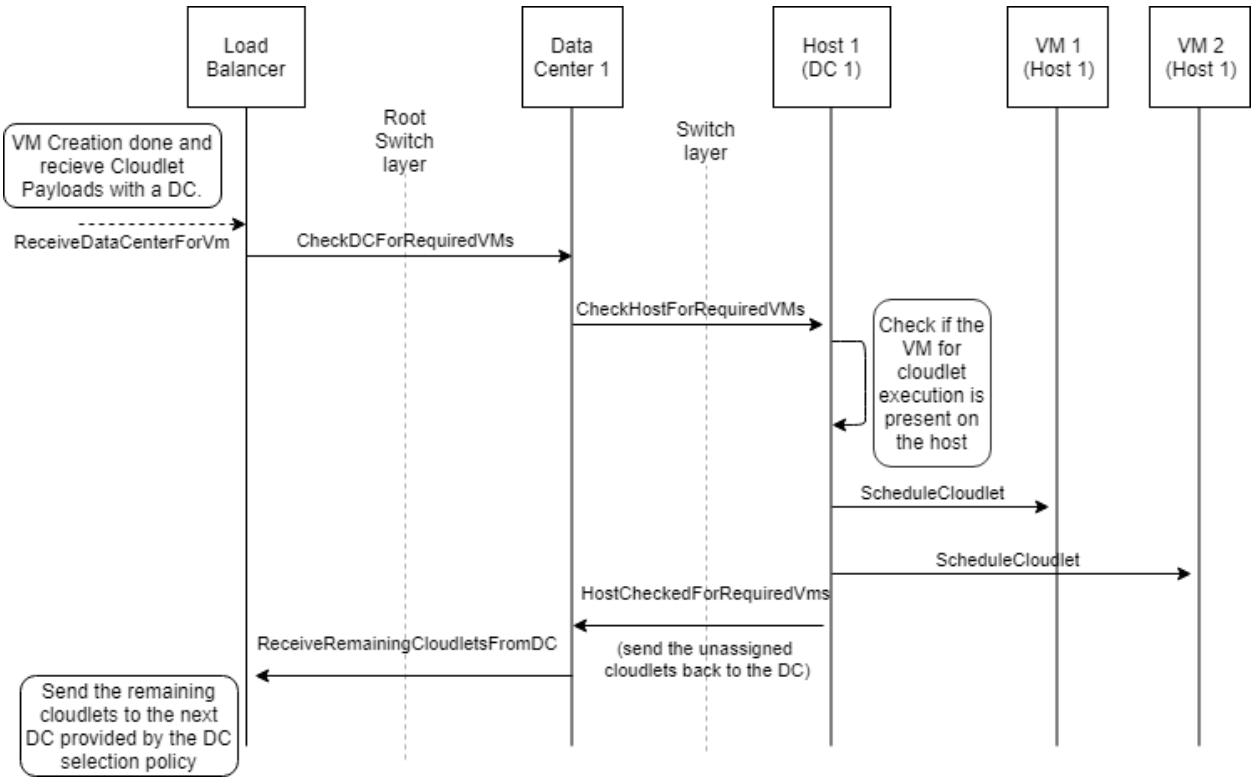

Figure 6: Cloudlet Assignment on the VM.

The user provides with the cloudlet payload which needs to be assigned on the VM. The VM identification is provided in the cloudlet payload.

Once the Load Balancer receives the request for cloudlet assignment, it asks for the Data Centers list from the Cloud Information Service (CIS). The list of cloudlets is a passed to the selected Data Center (given by Data Center Selection Policy) and subsequently to all its hosts. The host maintains the list of VMs currently running on it. Each cloudlet is checked if it's required VM is present on the current host. If the required VM is present, then the cloudlet will be sent for its execution. All the unassigned cloudlets are sent back to the Data Center and subsequently to the Load Balancer. The Data Selection Policy provides the next Data Center to the Load Balancer for searching the VMs for the unassigned cloudlets. This process is repeated either until all the cloudlets are assigned to the VMs or no more Data Centers are left to be searched.

3.1.4. Scheduling

Scheduling is the process of assigning CPU cores to VMs and eventually to cloudlets for execution that represents the execution of applications.

3.1.4.1. Time Actor

In the Cloud Simulator the entire infrastructure and execution is a simulation of the real-world cloud system. Naturally, the concept of time needs to be simulated too. Otherwise, the cloudlets will execute for the same length of time as real-world applications. While the simulator can execute cloudlets for the same amount of time as the real-world applications, that may not be necessary for testing most characteristics of the system. In order to incorporate the simulated time, we need an entity (Time actor) that manages the time. All other actors will receive time information from this actor. This will ensure that all entities within the system will be synchronously executed.

The Time Actor [5] is created once the entire infrastructure is created within the simulation and it's time to schedule VMs and cloudlets. Upon creation, it obtains the datacenter list from the CIS and sends a Time slice to each datacenter. A datacenter propagates this message to the actors under its hierarchy (hosts, VMScheduler and VMs). Thus, all resources in the system receive the time slice.

This time slice represents a fixed amount of time. A host on receipt of the time slice schedules VMs for the total time period. The VM on receipt of a portion of the time slice schedules cloudlets within it. Thus, the cloud system continues execution until the time slice is complete.

Upon completion of execution each VM sends a *TimeSliceCompleted* message to its host. Each host, upon receipt of the *TimeSliceCompleted* message from all its VMs propagates the message to its datacenter. Similarly, each datacenter, upon receipt of the *TimeSliceCompleted* message from all its hosts propagates the message to the Time actor. The Time actor waits till all the

datacenters send the *TimeSliceCompleted* message after which it sends a new time slice in the same fashion as described above.

In case the time slice has expired, and the cloudlet has not completed its execution, it waits for the next time slice. The Time Actor sends the new time slice on receiving *TimeSliceCompleted* message from all datacenters. The process goes on until no cloudlets are pending execution.

The layered propagation of time messages ensures that the Time actor doesn't cause bottleneck in the system [5]. Although the number of messages passed is higher, the penalty is distributed across the system.

Figure 6 shows how the time information is passed from the Time actor to the datacenter and the reverse-propagation of *TimeSliceCompleted* message. Following this the Time actor initiates the next time slice.

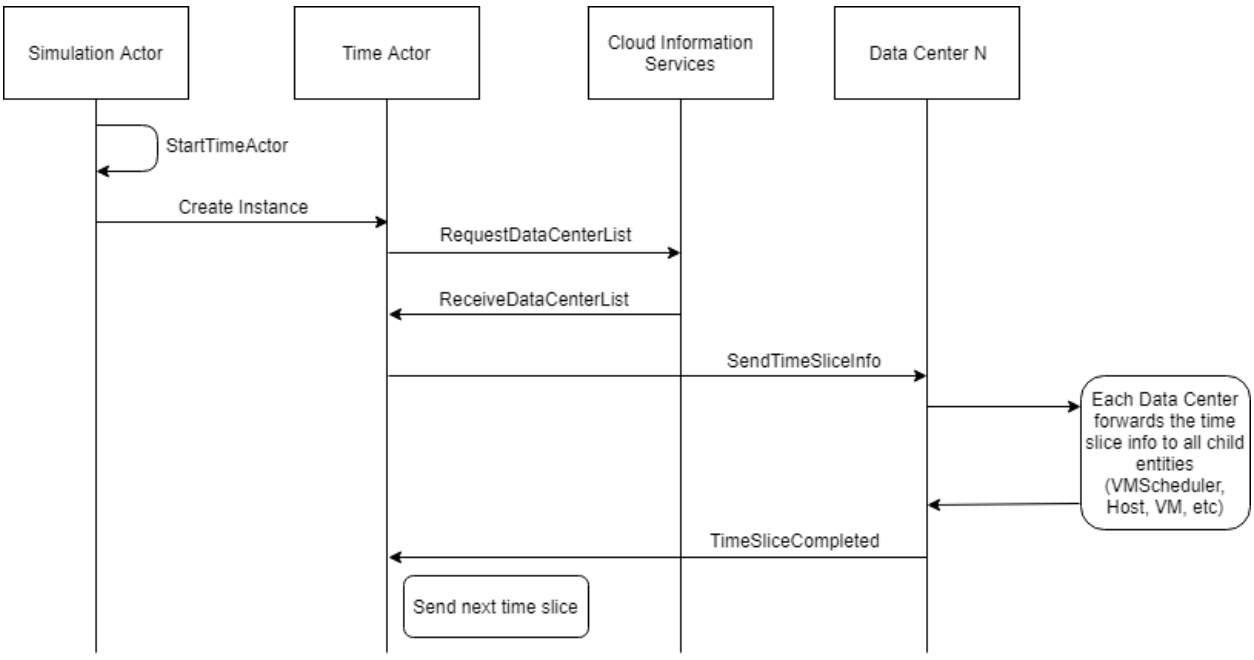

Figure 7: Time Actor and propagation of time information.

3.1.4.2. VM Scheduling

The VM Scheduler Policy is responsible to assign processing elements and time slices to VMs within a host. Every host has a VM Scheduler child actor. Thus, the VM Scheduling policy is implemented at the host level and each host can define this policy uniquely. The implementation is injected as a dependency to the VM Scheduler actor during creation based on the configuration provided by the user. Currently, the framework provides two implementations of the policy- Time Shared VM Scheduler and Space Shared VM Scheduler. The user can inject their own implementation for VM Scheduler.

When the Host receives the time slice, it instructs the VM Scheduler actor to schedule the VMs within the available resources at the host. The VM Scheduler actor requests each VM (child actor of the host) within the host to send its resource requirement. On receiving the resource requirement from all VMs, the scheduler decides the VM scheduling strategy based on the injected VM Scheduling Policy. For instance, the Time-Shared policy equally divides the time slice among the competing VMs. The Space-Shared policy divides the processing elements (cores) among the VMs based on the requirement and each VM gets the entire amount of time. Once the scheduling policy for each VM is decided, the scheduler sends the respective time slice to each VM actor. The VM actor performs its execution within the amount of resources and time awarded to it and sends a *TimeSliceCompleted* message back to the VM Scheduler that signals the completion of the awarded time. The scheduler then waits for the arrival of the next time slice after which, it repeats the scheduling process with the remaining VMs.

The VM Scheduler Policy is responsible to assign processing elements and time slices to VMs within a host. Every host has a VM Scheduler child actor. Thus, the VM Scheduling policy is implemented at the host level and each host can define this policy uniquely. The implementation is injected as a dependency to the VM Scheduler actor during creation based on the configuration provided by the user. Currently, the framework provides two implementations of the policy- Time

Shared VM Scheduler and Space Shared VM Scheduler. The user can inject their own implementation for VM Scheduler.

When the Host receives the time slice, it instructs the VM Scheduler actor to schedule the VMs within the available resources at the host. The VM Scheduler actor requests each VM (child actor of the host) within the host to send its resource requirement. On receiving the resource requirement from all VMs, the scheduler decides the VM scheduling strategy based on the injected VM Scheduling Policy. For instance, the Time-Shared policy equally divides the time slice among the competing VMs. The Space-Shared policy divides the processing elements (cores) among the VMs based on the requirement and each VM gets the entire amount of time. Once the scheduling policy for each VM is decided, the scheduler sends the respective time slice to each VM actor. The VM actor performs its execution within the amount of resources and time awarded to it and sends a TimeSliceCompleted message back to the VM Scheduler that signals the completion of the awarded time. The scheduler then waits for the arrival of the next time slice after which, it repeats the scheduling process with the remaining VMs.

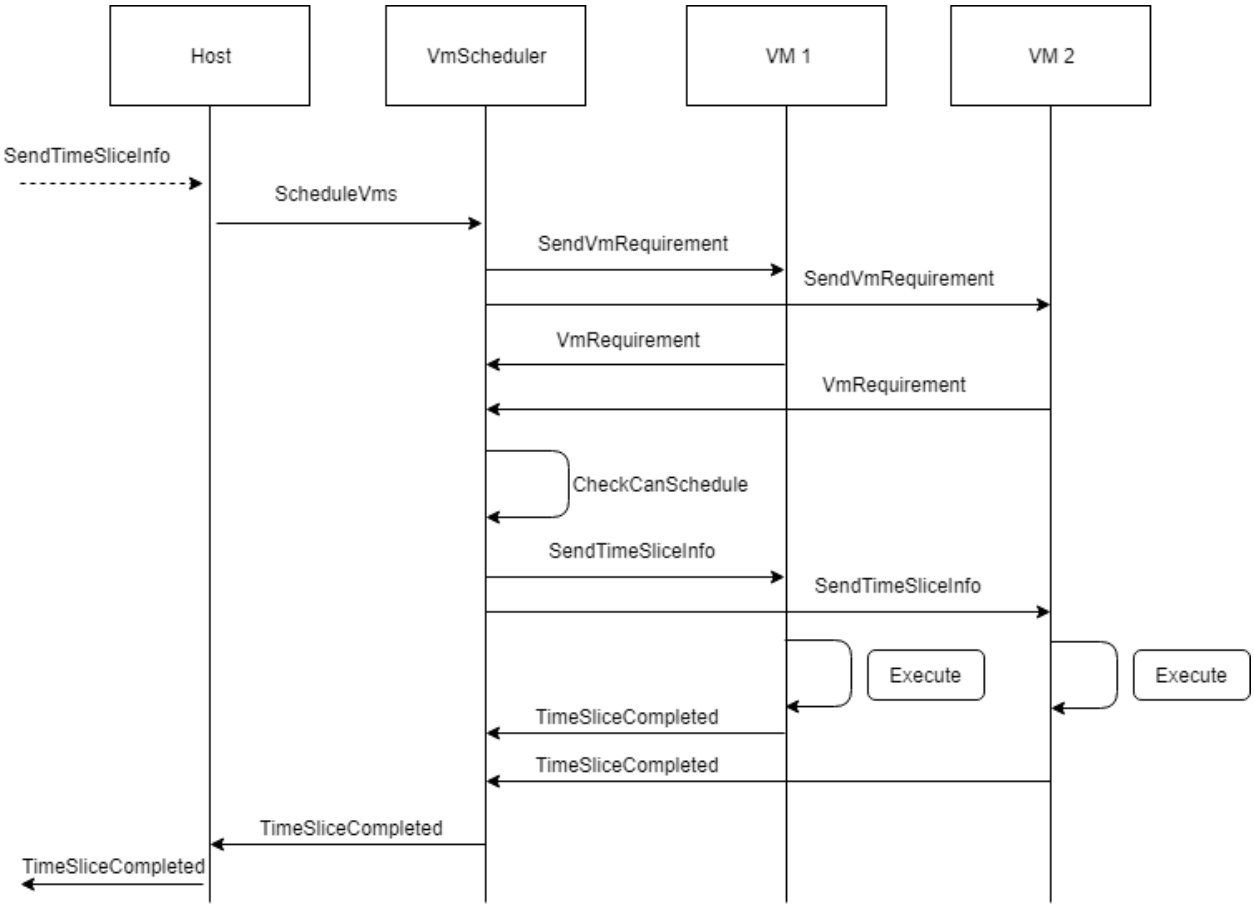

Figure 8:Sequence of message flow depicting VM scheduling.

3.1.4.3. Cloudlet Scheduling

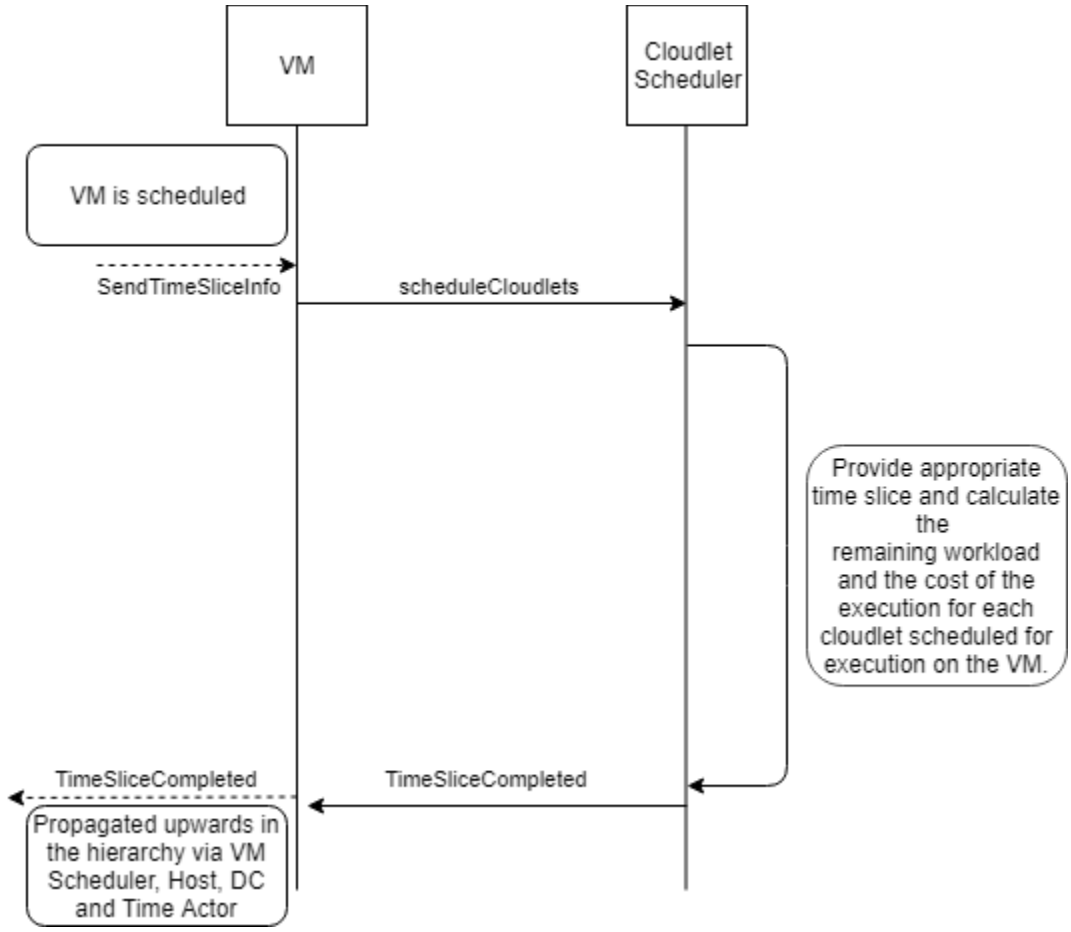

Figure 9: Cloudlet scheduling with the arrival of Time Slice.

Once the VM receives the time slice from the VM Scheduler, it provides it to the cloudlet scheduler. The cloudlet scheduler, depending on the implemented scheduling policy, calculates the remaining payload and the cost of execution. A Time-Shared Policy would divide the time slice into equal intervals based on the number of cloudlets to be executed. A new list of cloudlets with the updated values for cost, remaining payload, and if the cloudlet is completed then its current system start and end time is also provided.

3.2. Switch

In the Cloud Simulator, there is a huge scope of inter-host communication and inter-VM communication. To handle this, it is not feasible to store the information of every host/VM at every other host/VM. Thus, using switches makes the communication between indirectly related entities feasible.

All messages passing through switches in the Cloud Simulator extend the *NetworkPacket* type. *NetworkPacket* provides a parameter *NetworkPacketProperties* that is used to store sender and receiver information for a message.

We also add a delay component to every switch to represent the latency of processing a message. Thus, delay is added to every cloudlet which will be accounted for when the cloudlet execution time is computed. We define 3 types of switches in the Cloud Simulator- Root Switch, Aggregate Switch and Edge Switch.

3.2.1. Root Switch

Connects the datacenters with entities outside the cloud infrastructure. Figure 9 shows the Root Switch accept Network Packets from external entities like CIS and Load Balancer and forward it to the intended datacenter. Similarly, it also sends the packets sent by the datacenters to the intended external entity.

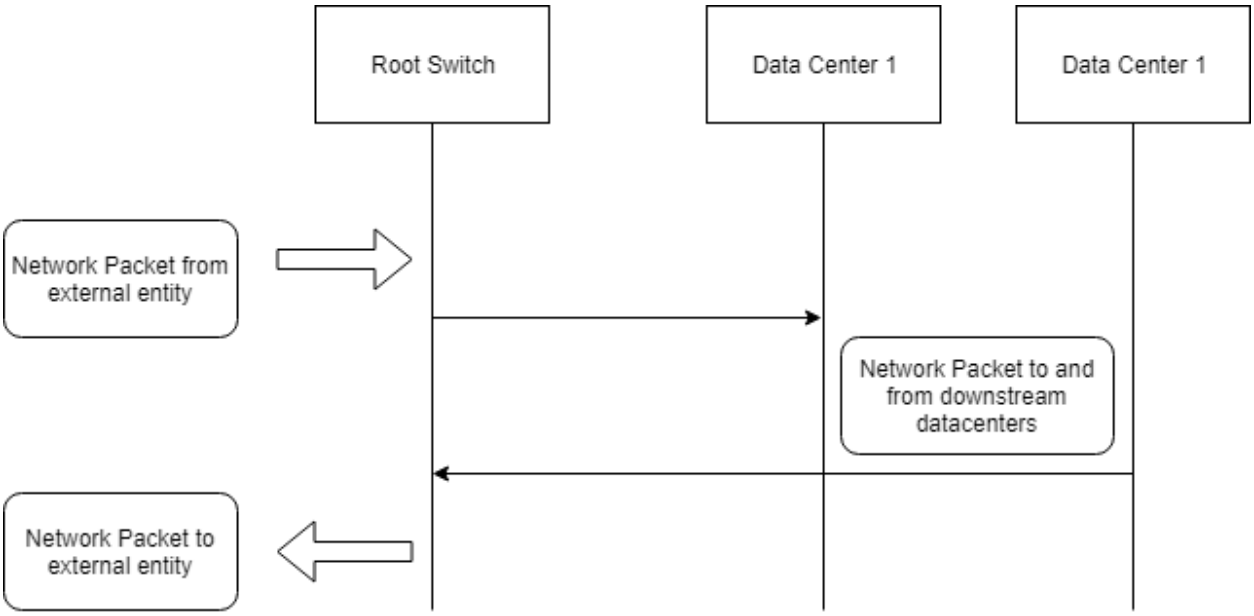

Figure 10: Root Switch communication specification.

3.2.2. Edge Switch

The edge switch is connected to another switch or datacenter in the upstream and is connected to hosts in the downstream. Figure 10 shows the edge switch accept message from the upstream/downstream, resolve the destination and send it forward in the downstream/upstream.

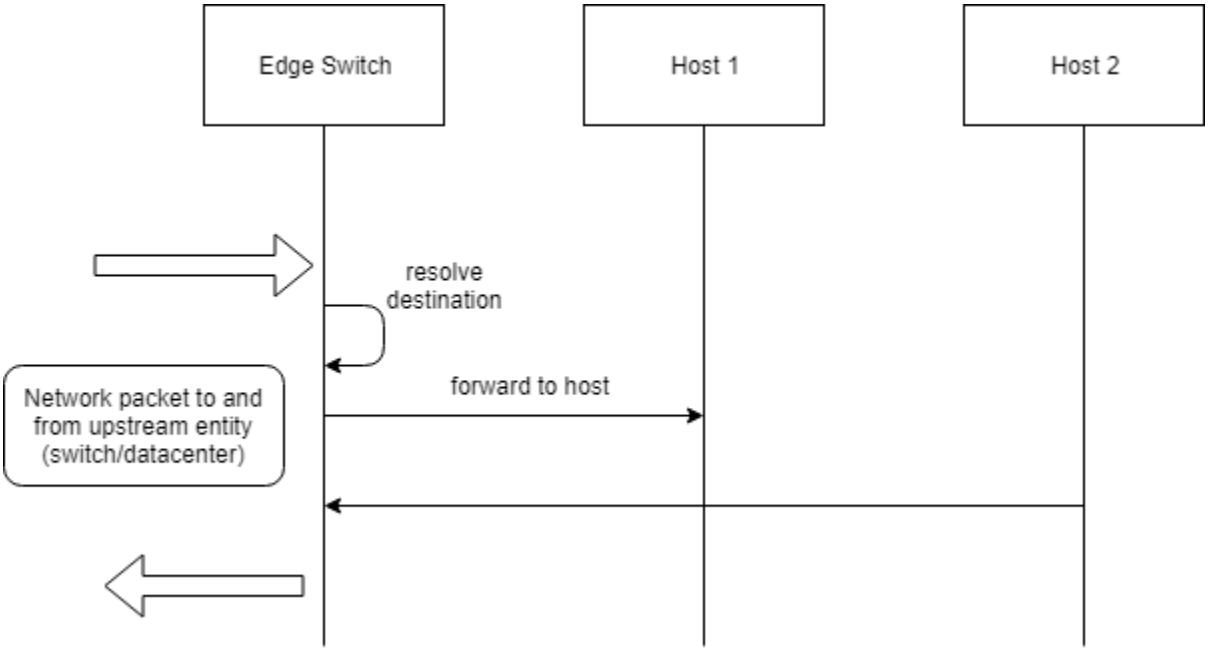

Figure 11: Edge Switch communication specification.

3.2.3. Aggregate Switch

The aggregate switch is connected to another switch or datacenter in the upstream and is connected to another switch in the downstream. Figure 10 shows the edge switch accept message from the upstream/downstream and send it forward in the downstream/upstream.

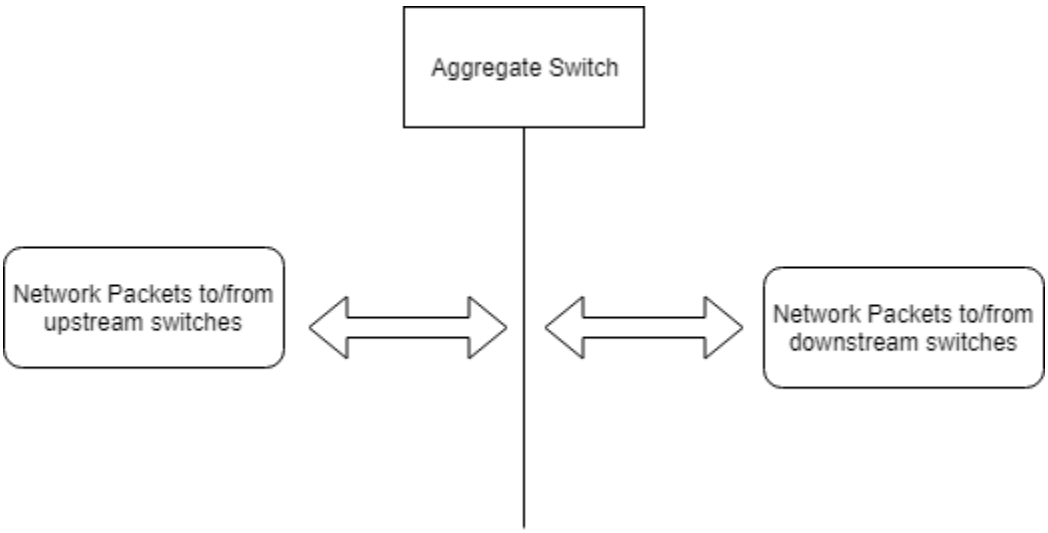

Figure 10: Aggregate Switch communication specification.

4. EXPERIMENTS

In this section we present the experiments performed on the system. The experiments were conducted on a Windows 10 machine that had 8GB of RAM and an Intel i5-7200U processor which is a dual core processor with 2 threads per core. Also, the processor has a base frequency of 2.50 GHz. For the experiments, the JVM was provided with a maximum of 4GB memory.

3.4. Scalability and overhead evaluation

This experiment is aimed at analyzing the memory usage and scalability of the simulator for different activities and different sizes of load. In the first experiment we test the infrastructure setup process. While it is true that the infrastructure consists of various entities, we measure the size of the load by the number of hosts present in the system. We perform the experiment in two scenarios; with all hosts created at a single datacenter and with hosts divided between two datacenters. We present the time and memory consumption statistics for the number of hosts varying from 10k to 1M. The final reading for 1M hosts is performed to test the system at extreme load conditions. All the data points have been gathered by making 10 observations and averaging them.

Figure 21 shows the amount of time required to setup the cloud infrastructure as a function of number of hosts. The two lines represent the equations for experiments with 1 datacenter and 2 datacenters respectively. The time taken to setup 1M hosts (not in the chart) was 37.16s and 38.71s with 1 datacenter and 2 datacenters respectively.

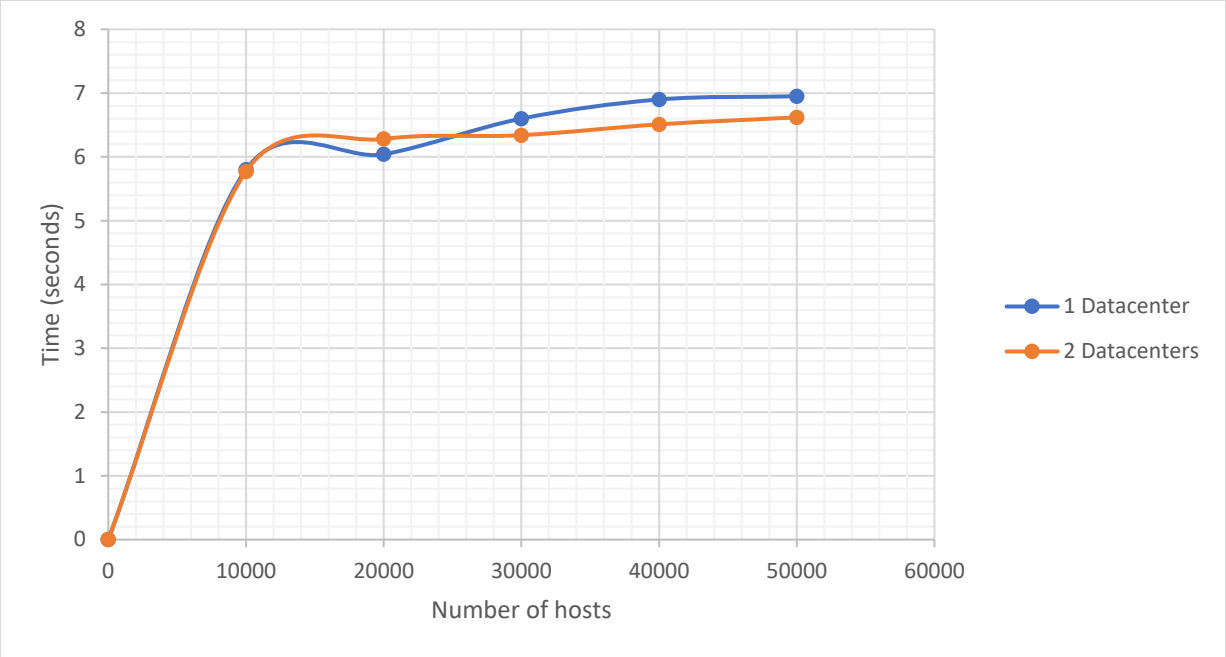

Figure 11: Time taken to setup hosts for different datacenter settings.

Figure 22 shows the memory consumption statistics for the same experiment as above. The amount of memory required for the execution is represented as a function of number of hosts. Once again, we see that the function is sublinear. The amount of memory required to setup 1M hosts (not in the chart) was 1995MB and 2195MB with 1 datacenter and 2 datacenters respectively.

The results show that the functions are sublinear and thus the overhead of infrastructure setup doesn't grow linearly with the number of hosts in the system. These experiments prove that the Cloud Simulator can be used to setup large-scale simulation environments with minimum overhead in terms of time and memory consumption.

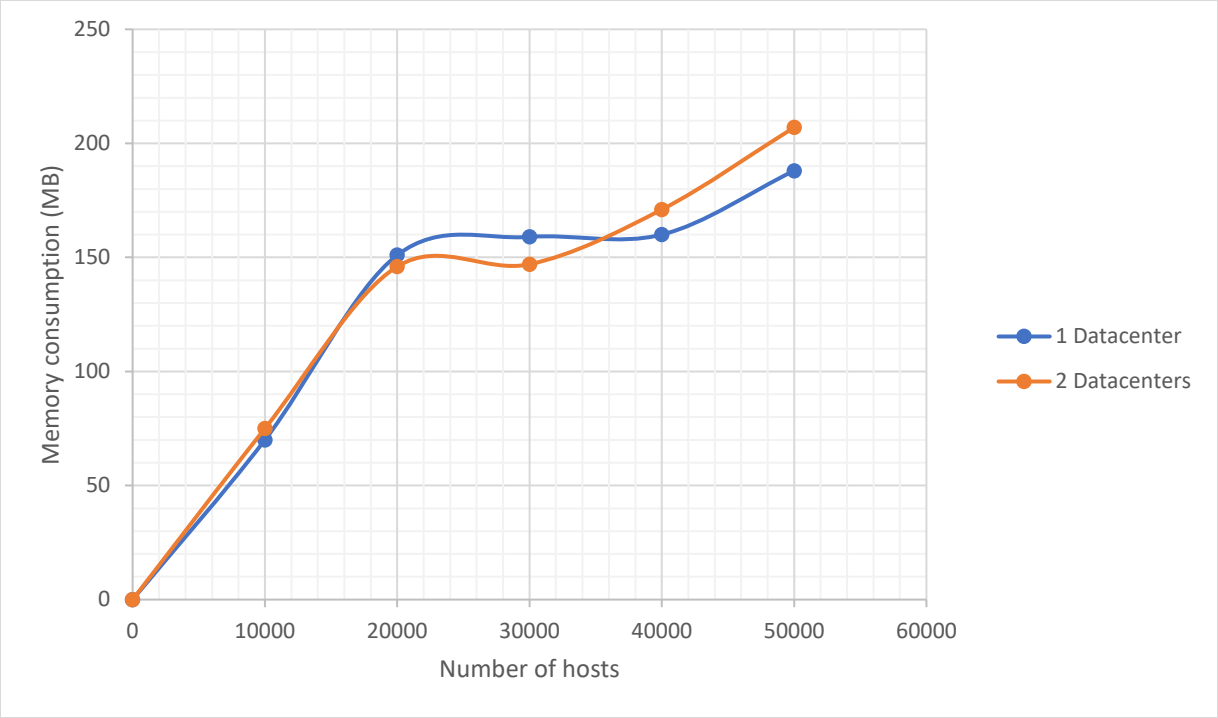

Figure 12: Memory consumption for setup of hosts for different datacenter settings.

The infrastructure setup process mainly reflects the overhead of creating actors in the system along with relatively smaller overhead of message passing between the actors. However, the execution of the simulator heavily involves complex interactions between actors. To properly analyze the performance of the simulator we perform experiments involving allocation of virtual machines, scheduling of virtual machines and execution of cloudlets in the system.

For this experiment, we fix the number of datacenters, hosts and virtual machines in the system and keep on varying the number of cloudlets deployed. As the number of cloudlets increases under fixed resources, the number messages passed in the system increase and so does the overhead of scheduling and execution.

Charts below show the result of experiments for a setup with 1 datacenter, 1k hosts, 1k virtual machines and varying number of cloudlets from 1k - 5k. Figures 23 shows execution time of the

simulator as a function of the number of cloudlets deployed in the system and Figure 24 shows the memory consumption of the execution as a function of the number of cloudlets. As can be seen, the increased overhead of message passing has impacted the time and memory functions from the previous experiment. However, the maximum time and memory usage is well within the acceptable limits for large workloads.

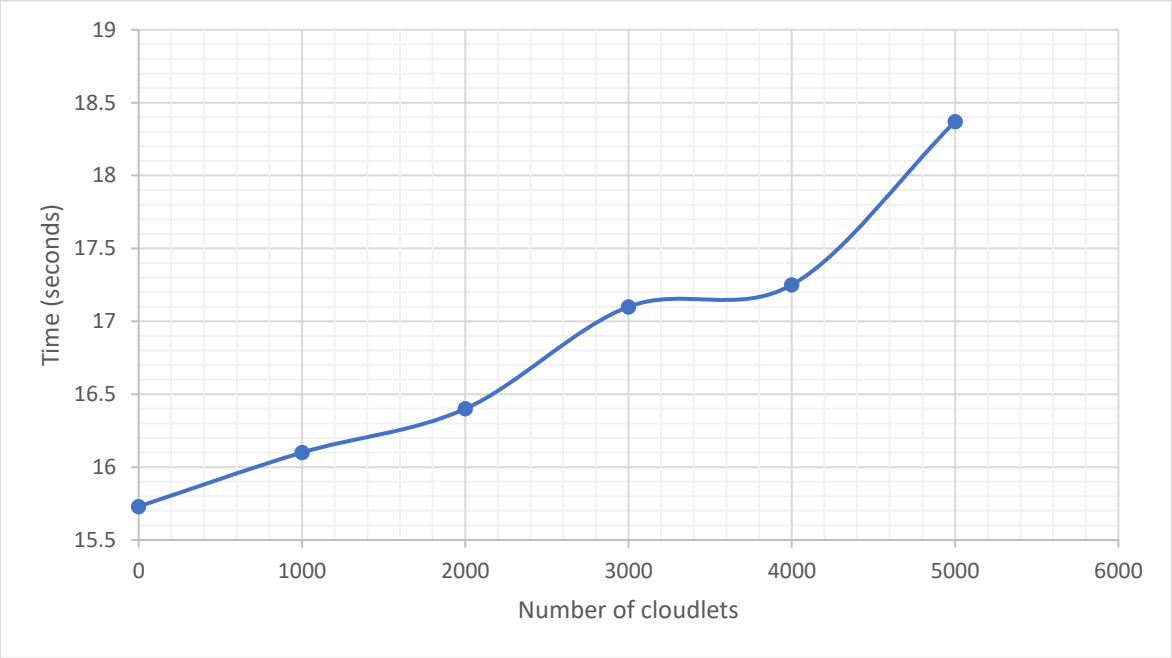

Figure 13: Time consumption for varying number of cloudlets.

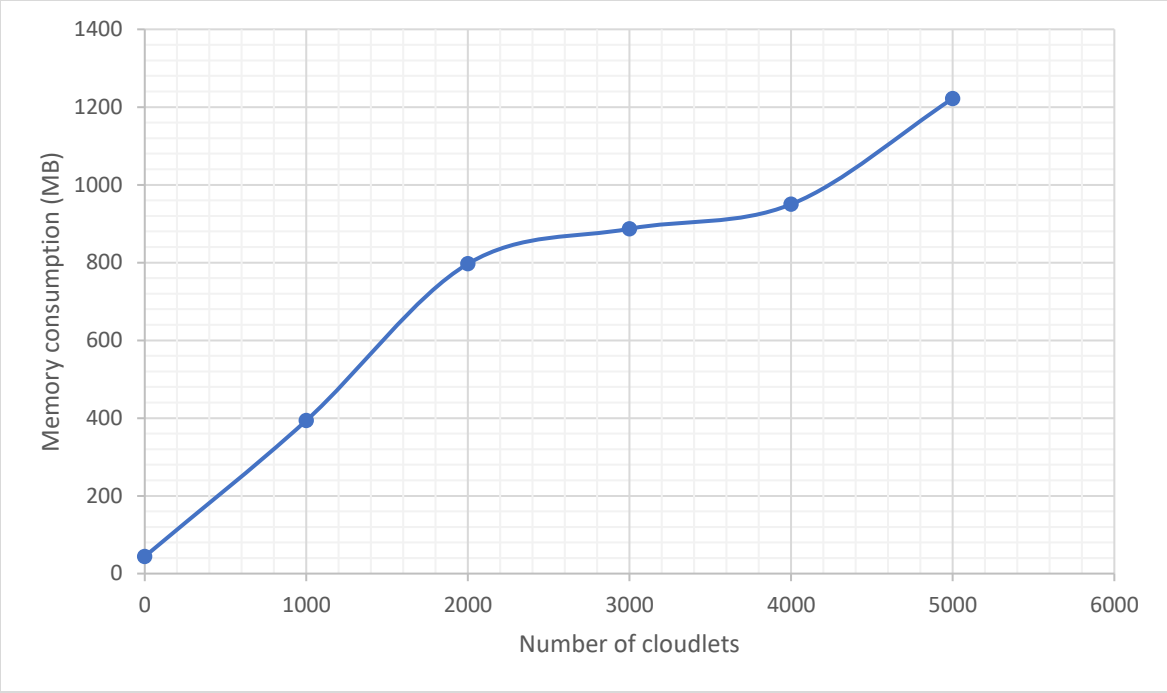

Figure 14: Memory consumption for varying number of cloudlets.

5. FUTURE WORK

Various cloud providers implement payment schemes like on-demand, reserved instances, spot instances and dedicated hosts. Integrating these payment schemes into the simulator would improve the calculation of the cost of execution.

Data Center Selection Policy is implemented to select the first DC from the list of DCs returned by the Cloud Information Service (CIS). Better selection policies based on the availability zones of the data centers can be implemented to reduce the search time of find the right DC for the payloads.

Currently, the cloudlet and VM payload is provided statically i.e. before the start of the simulation. Functionality of dynamic payload can be configured within the simulator; providing cloudlet and VM payload as and when they become available.

6. CONCLUSION

To test various types of cloud policies (allocation, scheduling, cost calculation, etc.) cloud providers/researchers need systems that can be used to evaluate those policies. Thus, there is a need for a simulator that can be configured to represent a cloud infrastructure of required configuration.

To meet these requirements, we have successfully implemented a cloud simulator using the Akka actor system. We have demonstrated through experiments that the system is stable for different sizes of workload. Also, it can easily scale up with acceptable time and memory overheads.

The simulator leverages the advantages of Akka's actor model. We have noted how the use of Akka's actor model helps us in avoiding the traditional concurrency issues and enforces immutability through the framework. This allows us to easily predict the behavior of the simulation system. It also gives developers the freedom to extend the system in order to adapt to their experimental settings without worrying about low-level concurrency issues.

Using this simulation tool, the cloud providers can easily test their real-world infrastructures and remove performance bottlenecks before laying the physical system. Also, they can easily test their policies to ensure better Quality of Service (QoS) when the infrastructure is deployed.

7. REFERENCES

[1] L. S. B. V. M Odersky, Programming in scala, San Fracisco: Artima Press, 2008.

[2] M. Odersky, The Scala Language Specification [Version 2.9], Programming Methods Laboratory EPFL..

[3] M. Thurau, "Akka framework," *University of Lübeck,* 2012.

[4] R. Buyya, R. Ranjan and R. N. Calheiros, "Modeling and Simulation of Scalable Cloud Computing Environments and the CloudSim Toolkit: Challenges and Opportunities," *2009 International Conference on High Performance Computing & Simulation, Leipzig,* pp. 1-11, 2009.

[5] H. A. Waite, "ScaNS: Using Actors for Massive Simulation of Distributed Routing Algorithms," 2013.

[6] R. R. A. B. C. A. F. D. R. R. B. Rodrigo N. Calheiros, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software—Practice & Experience,* pp. 23-50, 2011.

[7] "Pure Config," [Online]. Available: https://github.com/pureconfig/pureconfig. [Accessed 2019].

[8] "Human-Optimized Config Object Notation (HOCON)," [Online]. Available: https://github.com/lightbend/config/blob/master/HOCON.md#hocon-human-optimized-config-object-notation. [Accessed 2019].

[9] A. documentation, "How the Actor Model Meets the Needs of Modern, Distributed Systems [Version 2.5.22]," [Online]. Available: https://doc.akka.io/docs/akka/current/guide/actors-intro.html.

[10] W. A, "Computing in the clouds," *NetWorker,* 2007.

[11] "Props for a Akka Actor," Akka Documentation, 2019. [Online]. Available: https://doc.akka.io/api/akka/current/akka/actor/Props.html.