National College of Ireland

Configuration Manual

MSc Research Project
Cloud Computing

Arun Kumar Dasari
Student ID: X20155361

School of Computing
National College of Ireland

Supervisor: Aqeel kazmi

National College of Ireland
Project Submission Sheet
School of Computing

National College of Ireland

Student Name:	Arun Kumar Dasari
Student ID:	X20155361
Programme:	Cloud Computing
Year:	2021
Module:	MSc Research Project
Supervisor:	Aqeel kazmi
Submission Due Date:	16/12/2021
Project Title:	Configuration Manual
Word Count:	975
Page Count:	6

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard speci ed in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Arun Kumar Dasari
Date:	16th December 2021

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator o ce must be placed into the assignment box located outside the o ce.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Arun Kumar Dasari
X20155361

1 Introduction

1.1 Purpose of the document

The NCI research study set of rules was followed in the design of this manual. The application, methods, and tools employed in this project are detailed in this document. This is done by using spring boot microservices . In these services each services is for speci c purpose. In addition we have implemented Apache Kafka messaging services to get the utilization status of virtual machines.

2 System Requirements

The system requirements to run the project

- Operating System: Windows 10
- System Processor: Intel i5 core CPU @ 2.30GHz
- RAM: 4 GB
- System type: 64-bit operating system, x64-based processor

3 Installations

3.1 JAVA installation

It is possible to simulate the cloud environment using the Eclipse IDE with an integrated JAVA development plugin. Installing it is as simple as following these rst few steps:

- Install the Java Development Toolkit (JDK), which is required because the simulation is written in the Java programming language. https://dist.springsource.com/release/STS/ind
- Install the spring tool suite IDE to run https://spring.io/tools3/sts/all
- Open STS IDE by unzipping the downloaded folder and running the executable le (.exe).

3.2 Cloudsim Plus Installation:

CloudSim Plus is a simulation framework that is used for simulating the cloud computing environment.

The following are the second set of steps to implement the simulation in the Spring tool suite IDE:

- In maven repository and get the dependency of cloudsim plus .
- Just copy maven repository into pom.xml and then it added that dependency to the class path.

3.3 Apache Kafka Installation:

- Apache Kafka binaries will be used for the installation of Apache Kafka. Go the link https://kafka.apache.org/downloads
- After that create kafka folder and place the unzipped inside that folder.
- In the "con g/zookeeper.Properties" con guration le, change the path to the zookeeper data directory.

```
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
# the directory where the snapshot is stored.
dataDir=C:/kafka/zookeeper-data
# the port at which the clients will connect
clientPort=2181
# disable the per-ip limit on the number of connections since this is a non-production config
maxClientCnxns=0
# Disable the adminserver by default to avoid port conflicts.
# Set the port to something non-conflicting if choosing to enable this
admin.enableServer=false
# admin.serverPort=8080
```

Figure 1: Zookeeper con guration

- In the "con g/server.properties" le, change the Apache Kafka log le path.

```
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
# the directory where the snapshot is stored.
dataDir=C:/kafka/zookeeper-data
# the port at which the clients will connect
clientPort=2181
# disable the per-ip limit on the number of connections since this is a non-production config
maxClientCnxns=0
# Disable the adminserver by default to avoid port conflicts.
# Set the port to something non-conflicting if choosing to enable this
admin.enableServer=false
# admin.serverPort=8080
```

Figure 2: Kafka server con guration

4 Implementation

This sections about the implementation part of the proposed system. Initially, the implementation starts with Kafka zoo keeper followed by the creating zookeeper server where we need to execute to commands to run Kafka server. After that data center virtualization component is being started. After starting it is followed by the tracking of the resource service. Then for under loading purpose a block under load defection service is present and after under loading the overload detection is there. Finally the live virtual machine migration is obtained as the result of implementation of all these steps.

The Figure3 shows the execution of zoo keeper server which is the very beginning of the programming.

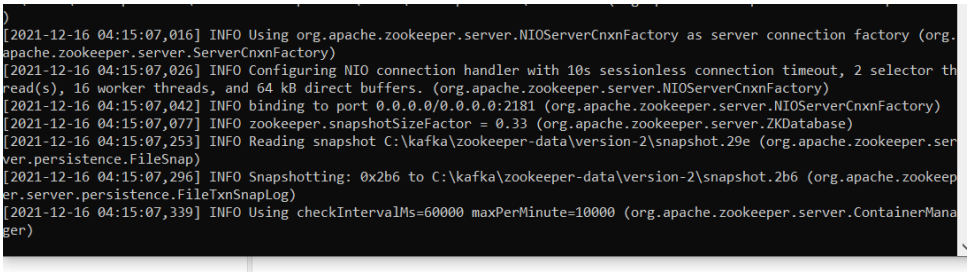

Figure 3: Zookeeper server

After this successful creation of zoo keeper server in a secured environment. Kafka server is to be initiated and for this following command is executed as shown in Figure 4.

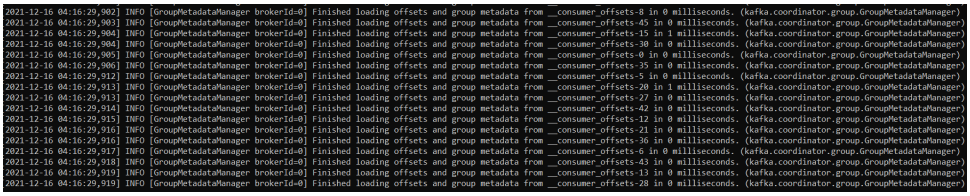

Figure 4: Kafka server

Figure 5 represents the code in which the simulation process is done in order to ensure the distribution of workload among cloudlets and later this data is sent to the Kafka consumer. So by all this we can say that the system is acting as the Kafka producer.

Tracking resource service is the next service which acts like a Kafka consumer and in which the data is taken from the Kafka producer in equal intervals of time. This is represented in the Figure 6.

In Figure 7 we can see how the overloaded services work. It's working is to take the data from tracking resource service in a particular interval of times and it intends to prepare list of overloaded services by considering the CPU utilization of the virtual machines.

Similarly in Figure 8 it is observed that how the under loaded services work the same way as or lead services like taking data from tracking resource services and making their list of under loaded services by considering the same parameter like CPU utilization.

Finally, by completing all these services like tracking research service, overloading and under loading services. the data is collected from all these resources and the migration of

```
for(Cloudlet cloud:cloudletList) {

          if(size==noOfCloudlets) {
              break;
          }
          else {
              finalCloudlets.add(cloud);
              size++;
              }

      }

      broker0.submitVmList(vmList);
      broker0.submitCloudletList(finalCloudlets);
      simulation.terminateAt(1000000);

      simulation.startSync();
      try {
          while(simulation.isRunning()){
              // tryDestroyVmAndResubmitCloudlets();

              simulation.runFor(INTERVAL);
              printVmCpuUtilization();
              DatacenterModel datacenterData=getDataCenterDetails();
              //datacenterData.setDatacenter(datacenter0);
              ProducerRecord<String, DatacenterModel> record=new ProducerRecord<String, DatacenterModel>("fourth_topic",datacenterData);
              producer.send(record);


          }
          producer.flush();
          producer.close();

      }
      catch(Exception e) {
          e.printStackTrace();

      }
```

Figure 5: Data Center Virtualization

```
public static DatacenterModel getDatacenterDeatils() {
    System.out.println("Subscribed to topic " + TOPIC);
    int i = 0;
    int counter=1;
    while (counter<=3) {
        ConsumerRecords<String, DatacenterModel> records = consumer.poll(100);
        for (ConsumerRecord<String, DatacenterModel> record : records) {

            // print the offset,key and value for the consumer records.
            /*
             * System.out.printf("offset = %d, key = %s, value = %s\n", record.offset(),
             * record.key(), record.value());
             */
            datacenterModel=record.value();



            for (final VMModel vm : record.value().getVmList()) {
                System.out.printf(" Vm %5d |", vm.getVmId());

            } System.out.println();

            for (final VMModel vm : record.value().getVmList()) {
                System.out.printf(" %7.0f%% |", vm.getVmCPUUtilization()*100);
            }


        }
        counter++;
    }

    return datacenterModel;
}
```

Figure 6: Tracking resources services

```
public List<HostList> getOverloadHostList(){
    HashMap<Integer, List<VMModel>> hostPerVm=new HashMap<Integer,List<VMModel>>(); // It contains map of host per VM
    List<HostList> overloadHostPerVM=new ArrayList<HostList>();

    for(VMModel vm:OverloadDetectionServicesApplication.datacenterModel.getVmList()) { // setting into to each vm to host as a key
        if(hostPerVm.containsKey(vm.getHostId())) {
            List<VMModel> vmListPerHost=hostPerVm.get(vm.getHostId());
            vmListPerHost.add(vm);
            hostPerVm.put(vm.getHostId(), vmListPerHost);
        }
        else {
            List<VMModel> vmListPerHost=new ArrayList<VMModel>();
            vmListPerHost.add(vm);
            hostPerVm.put(vm.getHostId(),vmListPerHost);
        }
    }

    for (Map.Entry<Integer, List<VMModel>> hostVMList : hostPerVm.entrySet()) { // computing average of each host
        List<VMModel> vmList=hostVMList.getValue();
        double sumOfCPUUtilization=0.0,averageCPUUtilization=0.0;
        for(VMModel vm:vmList) {
            sumOfCPUUtilization=sumOfCPUUtilization+vm.getVmCPUUtilization();
        }
        averageCPUUtilization=sumOfCPUUtilization/vmList.size(); // computing the average CPU utilization
        System.out.println("Average CPUUtilization of hostId"+hostVMList.getKey()+" is "+averageCPUUtilization+" with an number Of VMs: "+ vmList.size());
        if( averageCPUUtilization>=OverloadDetectionServicesApplication.slaContract.getCpuUtilizationMetric().getMaxDimension().getValue()) { // It will  check the whether the given host is over
            HostList overloadModel=new HostList();
            overloadModel.setAverageCPUUtilization(averageCPUUtilization);
            overloadModel.setHostid(hostVMList.getKey());
            overloadModel.setVmList(hostVMList.getValue());
            overloadHostPerVM.add(overloadModel);
        }
    }


    return overloadHostPerVM;
}
```

Figure 7: Overloaded Detection services

```
// This will throw UnderloadedHost
public List<HostList> getUnderloadHostList(){
    HashMap<Integer, List<VMModel>> hostPerVm=new HashMap<Integer,List<VMModel>>();
    List<HostList> underloadHostPerVM=new ArrayList<HostList>();


    for(VMModel vm:UnderloadedDetectionServicesApplication.datacenterModel.getVmList()) { // setting into to each vm to host as a key
        if(hostPerVm.containsKey(vm.getHostId())) {
            List<VMModel> vmListPerHost=hostPerVm.get(vm.getHostId());
            vmListPerHost.add(vm);
            hostPerVm.put(vm.getHostId(), vmListPerHost);
        }
        else {
            List<VMModel> vmListPerHost=new ArrayList<VMModel>();
            vmListPerHost.add(vm);
            hostPerVm.put(vm.getHostId(),vmListPerHost);
        }
    }

    for (Map.Entry<Integer, List<VMModel>> hostVMList : hostPerVm.entrySet()) { // computing average of each host
        List<VMModel> vmList=hostVMList.getValue();
        double sumOfCPUUtilization=0.0,averageCPUUtilization=0.0;
        boolean isHostActive=false;
        for(VMModel vm:vmList) {
            sumOfCPUUtilization=sumOfCPUUtilization+vm.getVmCPUUtilization();
        }
        averageCPUUtilization=sumOfCPUUtilization/vmList.size();
        if(averageCPUUtilization>0) {
            isHostActive=true;
        }

        System.out.println("Average CPUUtilization of hostId"+hostVMList.getKey()+" is "+averageCPUUtilization);
        //It will check the given host is underloaded or not
        if( isHostActive &&(averageCPUUtilization<UnderloadedDetectionServicesApplication.slaContract.getCpuUtilizationMetric().getMinDimension().getValue())) {
            HostList underloadModel=new HostList();
            underloadModel.setAverageCPUUtilization(averageCPUUtilization);
            underloadModel.setHostid(hostVMList.getKey());
            underloadModel.setVmList(hostVMList.getValue());
            underloadHostPerVM.add(underloadModel);
        }
```

Figure 8: UnderOverloaded Detection services

virtual machines takes place by using Ant colony optimization. And this is represented in the Figure 9.

```
for (int i = 0; i < noOfItterations; i++) {

     for (int ant = 0; ant < noOfAnts; ant++) {
         Set<Triple<HostList, VMModel, HostList>> currentAvailableTuples = new HashSet<>();
         currentAvailableTuples.addAll(finalTuples);
         Set<Triple<HostList, VMModel, HostList>> localMigrationPlaning = new HashSet<>();
         double localScoreing = 0;
         Set<VMModel> availableVms = new HashSet<>();
         availableVms.addAll(allVMList);
         Optional<Triple<HostList, VMModel, HostList>> optional = chooseNextTriple(currentAvailableTuples);
         Triple<HostList, VMModel, HostList> nextTriple;

         while (optional.isPresent()) {
             if (availableVms.isEmpty())
                 break;
             nextTriple = optional.get();
             currentAvailableTuples.remove(nextTriple);

             if (!availableVms.contains(nextTriple.getMiddle()) ) {
                 // System.out.printf("Skipped %s.\n", nextTriple.getRight());
                 optional = chooseNextTriple(currentAvailableTuples);
                 continue;
             }
             updateLocalPheromone(nextTriple);
             localMigrationPlaning.add(nextTriple);
            // nextTriple.getLeft().addVmMigratingOut(nextTriple.getMiddle());
             double score = getRouteScore(localMigrationPlaning);
             if (score > localScoreing) {
                 localScoreing = score;
                 VMModel vmMigrated = nextTriple.getMiddle();
                 availableVms.remove(vmMigrated);
                 // LOGGER.debug("Tuple: {}", nextTriple);
                 // LOGGER.debug("Score update: {}", localScore);
                 currentAvailableTuples.removeIf(tuple -> tuple.getMiddle() == vmMigrated);
             } else {
                 // LOGGER.info("Ant {} removing {}", ant, nextTriple);
                 localMigrationPlaning.remove(nextTriple);
             }
             // LOGGER.info("Migration plan: {}", localMigrationPlan);
             optional = chooseNextTriple(currentAvailableTuples);
```

Figure 9: Live VM Migration