

به نام خدا

محمد جواد ماهرالنقش - ۹۹۱۰۵۶۹۱

## گزارش پیاده سازی فاز ۱ پروژه سیستم های بی درنگ

### مقدمه

در این پروژه قصد داریم به کمک شبیه ساز iFogSim، چند الگوریتم زمانبندی را بر روی شبیه ساز iFogSim پیاده سازی کنیم. الگوریتم های FCFS، SJF، A<sup>2</sup>C و QLearning نیاز است تا در این شبیه ساز پیاده سازی شوند. سپس این شبیه سازی به ازای تعداد مقادیر مختلف مربوط به VM و Task انجام شود.

### صورت پروژه

قسمت های آبی رنگ برلی دو هفته آینده باید پیاده سازی شوند و قسمت سبز رنگ اهداف ما هستند که بر اساس آنها باید الگوریتم یادگیری تقویتی یا فراابتکاری را پیاده سازی کنید. قسمت Output تماماً در خود ifogsim وجود دارد در نتیجه نیازی به پیاده سازی نیست و صرفاً با جستجو میتوانید از توابع موجود بهره ببرید.

توجه: الگوریتم های فراابتکاری قسمت مقایسه که در فاز دوم نیاز به پیاده سازی دارند برلی آنها به شما رفرنس داده خواهد شد که از آن تا حدودی می توانید بهره ببرید.

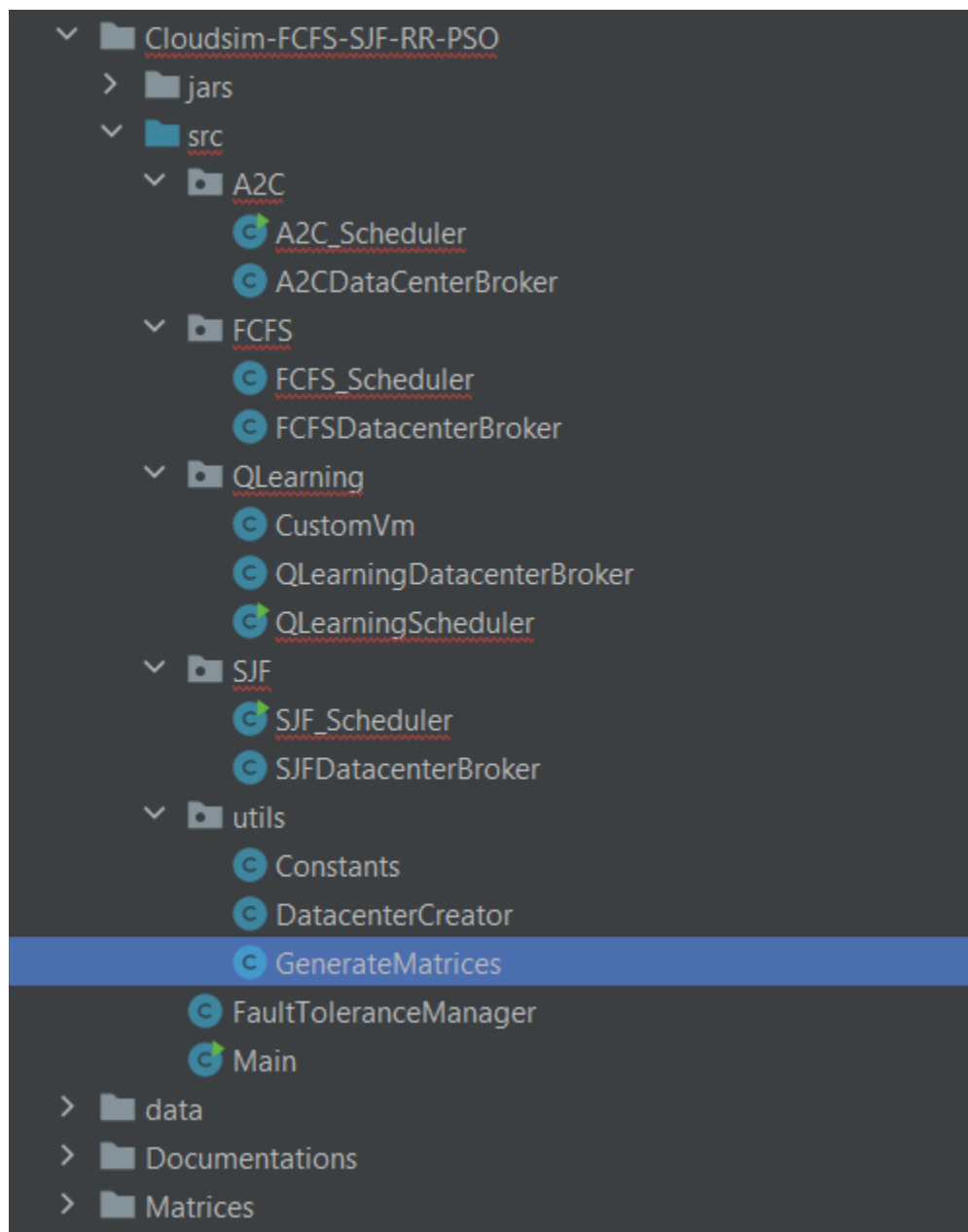
تعداد تسک (cloudlet): {۱۰، ۲۰، ۳۰، ۴۰، ۵۰} و {۲۰۰، ۴۰۰، ۶۰۰، ۸۰۰، ۱۰۰۰}

تعداد VM: {۲، ۵، ۸، ۱۰} و {۵۰، ۱۰۰، ۱۵۰، ۲۰۰}

نمونه خروجی excel نیز برلی تان در قالب فایل ارسال خواهد شد تا خروجی ها را مشابه آن ارسال کنید.

{task type: (soft real-time, periodic), task scheduling: [(q-learning, Advantage Actor-Critic (A<sup>2</sup>C)), (double q-learning)], Fault tolerance: (check pointing on jobs), resource allocation: (dynamic priority queue based on deadline), Outputs of first phase: (make span, completion time, wait time, response time, successful rate), Power: (-), Outputs of second phase: (CPU utilization, memory utilization, resource utilization, energy consumption), compare: (FCFS, SJF, Sarsa, Asynchronous Advantage Actor-Critic (A<sup>3</sup>C)), Objectives: (minimize the maximum (completion time, respose time))}

## ساختار کلی پروژه



ساختار کلی پروژه، مطابق با تصویری است که میبینید. یک فایل اصلی Main وجود دارد که الگوریتم ها را صدا زده و هر یک از ۴ الگوریتم در پوشه ای جداگانه پیاده سازی شده اند. سپس هر یک از این سیاست های پیاده سازی به ازای تعداد VM و تعداد Task های مختلف اجرا شده و نتایج خروجی در فایل هایی جداگانه ذخیره می شوند.

در ادامه به بررسی دقیق تر هر یک از توابع در پروژه می پردازیم.

- با توجه به یکسان یا مشابه بودن بسیاری از توابع، صرفا یکی از آنها به عنوان نمونه توضیح داده میشود.
- در نسخه های بعدی می توان نسبت به واحدسازی کدهای مشابه اقدام کرد، گرچه برخی از این توابع در همین نسخه نیز واحدسازی شده اند.
- تقریبا همه توابع دارای چند خط کامنت و توضیحات کلی هستند که نام تابع، کارکرد تابع، ورودی (ها) و خروجی (های) تابع را نشان می دهد.

## بررسی پیاده‌سازی

در ادامه به بررسی توابع و کدهای پیاده‌سازی شده می‌پردازیم.

- تابع `main` در کلاس `Main`

همانطور که در تصویر پایین می‌بینید، به ازای تعداد VM ها و Task های خواسته شده، هر بار `do_everything` صدا زده شده است.

مجموعاً  $8 \times 10 = 80$  بار این تابع صدا زده می‌شود.

```
// Author: Mohammad Javad Maheronnaghsh
// CloudSim and IFogSim
// Setup Java SDK version 16
// Last Update: July 1st
// Associated with Sharif University of Technology
// Professors: Dr. Mohsen Anasari, Dr. Sepideh Safari
// Supervisors: Abolfazl Younesi, Elyas Oustad

public class Main {
    public static void main(String[] args) {
        int[] tasks = {10, 20, 30, 40, 50, 200, 400, 600, 800, 1000};
        int[] datacenters = {2, 5, 8, 10, 50, 100, 150, 200};

        // Iterate over all possible number of task_n and datacenter_n
        for (int task : tasks) {
            do_everything(args);

            Constants.NO_OF_TASKS = task;
            for (int datacenter : datacenters) {
                Constants.NO_OF_DATA_CENTERS = datacenter;
                do_everything(args);
            }
        }
    }
}
```

- تابع `do_everything` در کلاس `Main`

همانطور که در تصویر پایین می‌بینیم، ابتدا به ازای ابرپارامترهای فعلی (تعداد `datacenter` ها و تعداد `cloudlet` ها)، پوشه‌ای با همین نام ایجاد می‌کنیم.

سپس `GenerateMatrices` را صدا زده تا دو فایل مربوط به `commMatrix` و `execMatrix` را ایجاد کند (هر یک از این ماتریس‌ها در پوشه‌ای متناسب با ابرپارامترها ذخیره می‌شود).

سپس ۴ الگوریتم زمانبندی بر روی اینها اجرا می‌شوند. و هر بار پس از اجرای الگوریتم‌ها، در پوشه‌ای متناسب با ابرپارامترهای فعلی، خروجی‌ها و ریزالت‌ها ذخیره می‌شوند.

```

/*
Function Name:
do_everything
Functionality:
execute all scheduling policies
input(s):
String[] args: this is the first input of the main function for start execution
output(s):
void: this function doesn't return anything directly, rather it saves the necessary data
to .csv files and also prints the log data
*/
private static void do_everything(String[] args) {
    String hyperparameters = "data/" + String.valueOf(Constants.NO_OF_DATA_CENTERS)
        .concat("-").concat(String.valueOf(Constants.NO_OF_TASKS));

    File directory = new File(hyperparameters);
    directory.mkdirs();

    new GenerateMatrices();

    // Execute the FCFS Scheduler
    FCFS_Scheduler.main(args);
    save_outputs(FCFS_Scheduler.getList(), FCFS_Scheduler.getExecMatrix(), FCFS_Scheduler.getCommMatrix(), hyperparameters.concat("/fcfs_data.csv"));

    // Execute the SJF Scheduler
    SJF_Scheduler.main(args);
    save_outputs(SJF_Scheduler.getList(), SJF_Scheduler.getExecMatrix(), SJF_Scheduler.getCommMatrix(), hyperparameters.concat("/sjf_data.csv"));

    // Execute the A2C Scheduler
    A2C_Scheduler.main(args);
    save_outputs(A2C_Scheduler.getList(), A2C_Scheduler.getExecMatrix(), A2C_Scheduler.getCommMatrix(), hyperparameters.concat("/a2c_data.csv"));

    // Execute the Q-Learning Scheduler
    QLearningScheduler.main(args);
    save_outputs(QLearningScheduler.getList(), QLearningScheduler.getExecMatrix(), QLearningScheduler.getCommMatrix(), hyperparameters.concat("/qlearning_data.csv"));
}

```

## • تابع save\_outputs در کلاس Main

وظیفه این تابع، ذخیره خروجی های خواسته شده در فایل های CSV متناظر است.

```

/*
Function Name:
save_outputs
Functionality:
save the outputs of all schedulers to .csv files
input(s):
List<Cloudlet> list: the list of all tasks (cloudlets)
double[][] execMatrix: execution time of cloudlets on different data centers.
double[][] commMatrix: communication cost between different cloudlets and data centers
String csvFilePath: filepath of the csv file (that has to be saved)
output(s):
void: this function doesn't return anything directly, rather it saves the necessary data
to .csv files
*/
public static void save_outputs(List<Cloudlet> list, double[][] execMatrix, double[][] commMatrix, String csvFilePath) {
    int size = list.size();
    Cloudlet cloudlet;

    // Create a CSV writer
    try (CSVWriter writer = new CSVWriter(new FileWriter(csvFilePath))) {
        String[] header = {"Cloudlet ID", "Status", "Data center ID", "VM ID", "Time",
            "Start Time", "Finish Time", "Waiting", "Completion", "Cost"};
        writer.writeNext(header);

        DecimalFormat dft = new DecimalFormat("####.##");
        dft.setMinimumIntegerDigits(2);

        double totalCompletionTime = 0;
        double totalCost = 0;
        double totalWaitingTime = 0;

        for (int i = 0; i < size; i++) {
            cloudlet = list.get(i);

            String cloudletId = dft.format(cloudlet.getCloudletId());
            String status = cloudlet.getCloudletStatus() == Cloudlet.SUCCESS ? "SUCCESS" : "Failure";
            String dataCenterId = dft.format(cloudlet.getResourceId());
            String vmId = dft.format(cloudlet.getVmId());
            String time = dft.format(cloudlet.getActualCPUTime());
            String startTime = dft.format(cloudlet.getExecStartTime());

```

- تابع calcMakespan در کلاس Main

وظیفه این تابع، محاسبه مقدار عددی پارامتر makespan بوده که از مواردی است که باید در خروجی ظاهر شوند.

```
/*
Function Name:
    calcMakespan
Functionality:
    calculates the makespan of a set of cloudlets executed on different data centers.
    The makespan is a metric that represents the total time taken to complete all the cloudlets in the simulation.
input(s):
    List<Cloudlet> list: the list of all tasks (cloudlets)
    double[][] execMatrix: execution time of cloudlets on different data centers.
    double[][] commMatrix: communication cost between different cloudlets and data centers
output(s):
    double makespan: the makespan of the cloudlets on the datacenters (that is a float number)
*/
private static double calcMakespan(List<Cloudlet> list, double[][] execMatrix, double[][] commMatrix) {
    double makespan = 0;
    double[] dcWorkingTime = new double[Constants.NO_OF_DATA_CENTERS];

    for (int i = 0; i < Constants.NO_OF_TASKS; i++) {
        int dcId = list.get(i).getVmId() % Constants.NO_OF_DATA_CENTERS;
        if (dcWorkingTime[dcId] != 0) --dcWorkingTime[dcId];
        dcWorkingTime[dcId] += execMatrix[i][dcId] + commMatrix[i][dcId];
        makespan = Math.max(makespan, dcWorkingTime[dcId]);
    }
    return makespan;
}
```

- تابع main در کلاس A2C\_Scheduler

در اینجا پیاده سازی تابع اصلی مربوط به زمانبند A2C را مشاهده میکنیم.

```
/*
Function Name:
    main
Functionality:
    run Q-Learning Scheduler
input(s):
    String[] args: Not important.
output(s):
    void: it doesn't return anything, it rather schedules the tasks based on the policy
*/
public static void main(String[] args) {
    Log.println("Starting A2C Scheduler...");

    execMatrix = GenerateMatrices.getExecMatrix();
    commMatrix = GenerateMatrices.getCommMatrix();

    try {
        int num_user = 1; // number of grid users
        Calendar calendar = Calendar.getInstance();
        boolean trace_flag = false; // mean trace events

        CloudSim.init(num_user, calendar, trace_flag);

        datacenter = new Datacenter[Constants.NO_OF_DATA_CENTERS];
        for (int i = 0; i < Constants.NO_OF_DATA_CENTERS; i++) {
            datacenter[i] = DatacenterCreator.createDatacenter("Datacenter_" + i);
        }
        A2CDataCenterBroker broker = createBroker("Broker");

        cloudletList = createCloudlet(broker.getId(), Constants.NO_OF_TASKS, idShift: 0);
        vmList = createVM(broker.getId(), Constants.NO_OF_DATA_CENTERS);

        broker.submitCloudletList(cloudletList);
        broker.submitVmList(vmList);

        CloudSim.startSimulation();

        List<Cloudlet> resultList = broker.getCloudletReceivedList();
        CloudSim.stopSimulation();
    }
}
```

## • تابع createVM

وظیفه این تابع، ایجاد ماشین های مجازی است. به عنوان پارامتر ورودی؛ تعداد ماشین های مجازی لی که باید تولید کند را گرفته و لیست ماشین های مجازی تولید شده را برمیگرداند.

```
/*
Function Name:
    createVM
Functionality:
    create virtual machine with the given userid and preferred general parameters
input(s):
    int userId: the userid related to this VM (this is not important that much!)
    int vms: number of virtual machines to be created
output(s):
    List<Vm> list: list of all created VMs
*/
private static List<Vm> createVM(int userId, int vms) {
    LinkedList<Vm> list = new LinkedList<>();

    long size = 10000; // image size (MB)
    int ram = 512; // VM memory (MB)
    int mips = 1000;
    long bw = 1000;
    int pesNumber = 1; // number of CPUs
    String vmm = "Xen"; // VMM name

    Vm[] vm = new Vm[vms];

    for (int i = 0; i < vms; i++) {
        vm[i] = new Vm(datacenter[i].getId(), userId, mips, pesNumber, ram, bw, size, vmm, new CloudletSchedulerSpaceShared());
        list.add(vm[i]);
    }

    return list;
}
```

## • تابع createCloudlet

وظیفه این تابع، ایجاد cloudlet یا همان تسک، به تعدادی است که در پارامتر ورودی میگیرد.

```
/*
Function Name:
    createCloudlet
Functionality:
    create cloudlets (tasks) with the given userid and preferred general parameters
input(s):
    int userId: the userid related to this VM (this is not important that much!)
    int cloudlets: number of cloudlets (tasks) to be created
    int idShift: Not important for this project!
output(s):
    List<Vm> list: list of all created VMs
*/
private static List<Cloudlet> createCloudlet(int userId, int cloudlets, int idShift) {
    LinkedList<Cloudlet> list = new LinkedList<>();

    long fileSize = 300;
    long outputSize = 300;
    int pesNumber = 1;
    UtilizationModel utilizationModel = new UtilizationModelFull();

    Cloudlet[] cloudlet = new Cloudlet[cloudlets];

    for (int i = 0; i < cloudlets; i++) {
        int dcId = (int) (Math.random() * Constants.NO_OF_DATA_CENTERS);
        long length = (long) (1e3 * (commMatrix[i][dcId] + execMatrix[i][dcId]));
        Cloudlet cl = new Cloudlet(dcId, idShift + i, length, pesNumber, fileSize, outputSize, utilizationModel, utilizationModel, utilizationModel);
        cl.setUserId(userId);
        cl.setVmId(dcId + 2);
        list.add(cl);
    }

    return list;
}
```

- سایر توابع پایه در کلاس های Scheduler

اینجا نیز ۴ تابع پایه که در همه کلاس های Scheduler استفاده شده اند را میبینید. تابع اول، Broker مربوطه را ایجاد میکند، تابع دوم، لیست cloudlet ها را برمیگرداند. تابع سوم و چهارم نیز ماتریس هایی که از کلاس GenerateMatrices ایجاد شده اند را برمیگرداند.

- توجه: در به روز رسانی های بعدی، میتوان دو تابع آخر را حذف کرد و آنها را از کلاس GenerateMatrices دریافت کرد.

```
/*
Function Name:
    createBroker
Functionality:
    create DataCenter Broker related to this scheduler
input(s):
    String name: name of the broker (arbitrary; it is not important that much)
output(s):
    A2CDataCenterBroker: a datacenter broker object
*/
private static A2CDataCenterBroker createBroker(String name) throws Exception {
    return new A2CDataCenterBroker(name, actorLearningRate: 0.1, criticLearningRate: 0.1, discountFactor: 0.1);
}

public static List<Cloudlet> getList() { return A2C_Scheduler.resultList; }

public static double[][] getExecMatrix() { return A2C_Scheduler.execMatrix; }

public static double[][] getCommMatrix() { return A2C_Scheduler.commMatrix; }
}
```

- تابع processCloudletReturn در کلاس های DatacenterBroker

این توابع وظیفه override کردن این تابع از کلاس پدر DatacenterBroker را دارند. ابتدا cloudlet را به getCloudletReceivedList اضافه کرده و سپس پردازش های مربوط به این الگوریتم را بر روی آن انجام میدهد.

```
/*
Function Name:
    processCloudletReturn
Functionality:
    Submit the cloudlet for execution on the selected VM
input(s):
    SimEvent ev:
output(s):
    void: it does not have output, rather it has to submit cloudlets (tasks)
*/
@Override
protected void processCloudletReturn(SimEvent ev) {
    Cloudlet cloudlet = (Cloudlet) ev.getData();
    getCloudletReceivedList().add(cloudlet);
    Log.println(CloudSim.clock() + ": " + getName() + ": Cloudlet " + cloudlet.getCloudletId() + " received");
    cloudletsSubmitted++;

    int vmId = cloudlet.getVmId();
    Vm vm = getVmList().stream().filter(v -> v.getId() == vmId).findFirst().orElse(null);
    // If the cloudlet (task) is executed, we should not enter it again
    if (vm != null && !cloudlet.isFinished()) {
        double reward = calculateReward(cloudlet);
        double[] state = calculateState(vm, cloudlet);

        // Update the actor and critic models
        updateModels(state, reward);

        // Select an action (using the actor model)
        int selectedVmId = selectAction(vm, state);
        cloudlet.setVmId(selectedVmId);

        // Submit the cloudlet for execution on the selected VM
        sendNow(getVmsToDatacentersMap().get(vm.getId()), CloudSimTags.CLOUDLET_SUBMIT, cloudlet);
    }
}
```

- توضیحات مربوط به AYC

به طور کلی هر استیت به کمک ۳ معیار مشخص میشوند:

۱. مقدار mips مربوط به vm

۲. زمان اجرای cloudlet

۳. اندازه فایل cloudlet

البته این فیچرها میتوانند کم و زیاد شوند و در صورت تمایل میتوانیم آنها را در تابع calculateState تغییر دهیم.

همچنین نیاز به تعیین پاداش یا همان reward داریم که به صورت  $\frac{1}{execTime}$  تعیین شده است. البته میتوانیم این تابع پاداش را به صورت های دیگری نیز تغییر دهیم و عوامل دیگری را نیز در نظر بگیریم (تغییرات لازم در تابع calculateReward میتوانند انجام شوند).

اکشن ها نیز به صورت vmId تعریف شده اند، یعنی اکشن وقتی X باشد یعنی vm شماره X را به این cloudlet اختصاص دهیم.

همچنین در AYC باید دومدل actor و critic داشته باشیم که میتوانند به صورت های مختلفی مثل hashmap، Q-Table و Neural Network باشند. میتوانیم مدل های دیگری از آنها را نیز برای پروژه های مشابه تست کنیم (برای این کار، باید همه جاهایی که از actorModel یا criticModel استفاده شده را به روزرسانی کنیم).

در تابع updateModels، در بخشی نیاز داریم که مقدار ارزش استیت بعدی را تخمین بزنیم. با توجه به اینکه نمیدانیم استیت بعدی چیست، به ازای تمام استیت های بعدی ممکن، میانگین میگیریم. این سیاست را نیز میتوان در این تابع (در صورت نیاز) تغییر داد.

- توضیحات مربوط به Q-Learning

پیاده سازی مطابق با الگوریتم مرسوم Q-Learning است. استیت های ما برابر با تسک های ما میشوند و اکشن های ما نیز تسک های بعدی میشوند.

رد صورت تمایل میتوانیم اکشن ها و استیت ها را با تغییر کلاس broker مربوط به آن تغییر دهیم.

- توضیحات مربوط به Check Pointing و Fault Tolerance

در تصویر پایین، پیاده سازی کلی مربوط به آن را میبینید که سعی شده کلیت CheckPointing پیاده سازی شود.

البته این صرفاً یک ProtoType اولیه است و از آن در کد استفاده نشده است و نیاز به تکمیل دارد.



```

// Author: Mohammad Javad Maheronnaghsh
// CloudSim and IFogSim
// Setup Java SDK version 16
// Last Update: July 1st
// Associated with Sharif University of Technology
// Professors: Dr. Mohsen Anasari, Dr. Sepideh Safari
// Supervisors: Abolfazl Younesi, Elyas Oustad

public class FaultToleranceManager {
    private static final String CHECKPOINT_FILE_PATH = "checkpoint.csv";

    public static boolean isCheckpointAvailable() {
        File checkpointFile = new File(CHECKPOINT_FILE_PATH);
        return checkpointFile.exists();
    }

    public static List<Cloudlet> loadCheckpointData() {
        try (FileInputStream fis = new FileInputStream(CHECKPOINT_FILE_PATH);
            ObjectInputStream ois = new ObjectInputStream(fis)) {
            return (List<Cloudlet>) ois.readObject();
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
        return null;
    }

    public static void saveCheckpointData(List<Cloudlet> list) {
        try (FileOutputStream fos = new FileOutputStream(CHECKPOINT_FILE_PATH);
            ObjectOutputStream oos = new ObjectOutputStream(fos)) {
            oos.writeObject(list);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void deleteCheckpointFile() {
        File checkpointFile = new File(CHECKPOINT_FILE_PATH);
        checkpointFile.delete();
    }
}

```

## خروجی ها

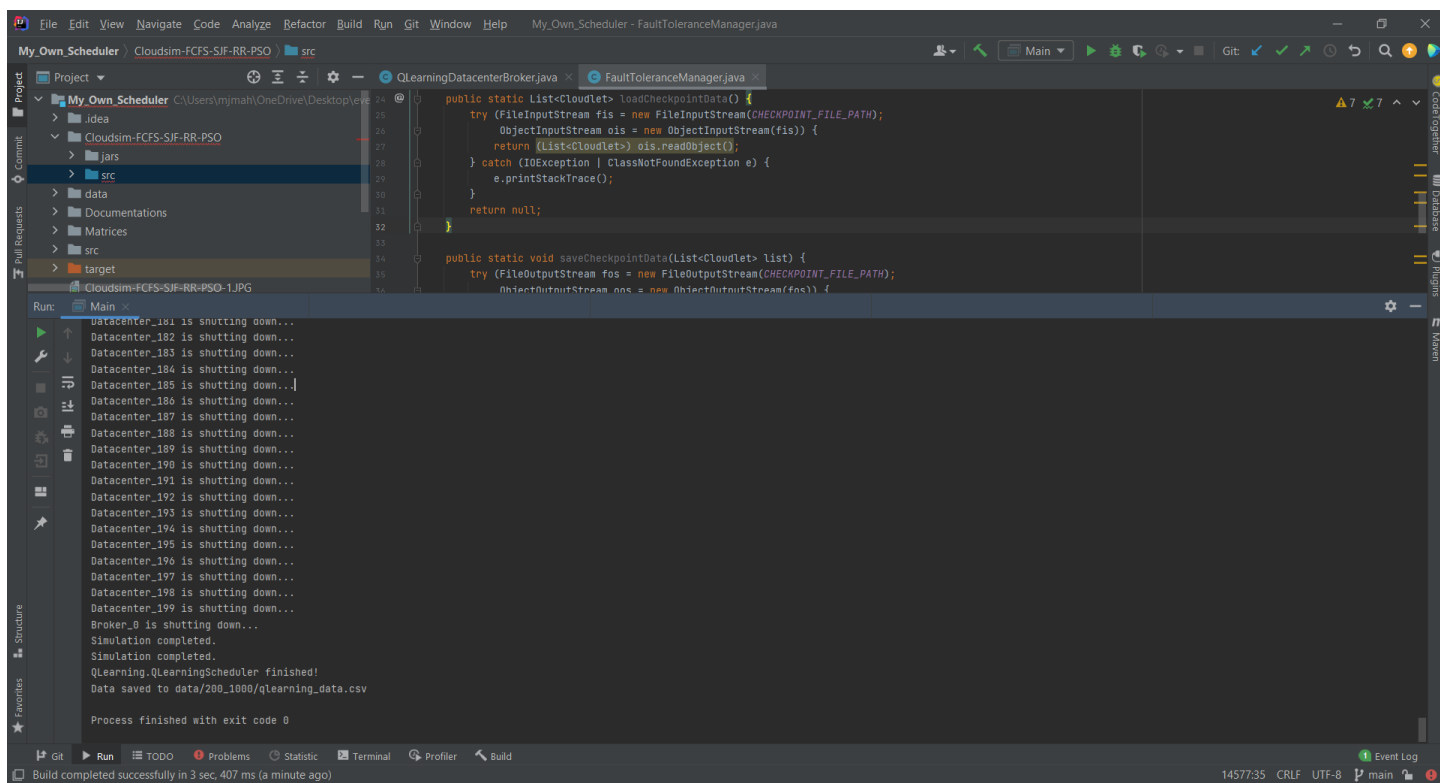
فایل های خروجی در پوشه ذخیره میشوند. میتوانید نمونه اجراهای قبلی را ببینید.

به ازای  $x$  تا Datacenter و  $y$  تا Cloudlet میتوانیم پوشه  $data/X\_Y$  را بررسی کنید که در آن ۴ فایل خروجی قرار داده شده است که هر کدام مربوط به یکی از ۴ سیاست زمانبندی است.

همچنین `commMatrix` و `execMatrix` متناظر با اینها را نیز میتوان در پوشه Matrices دید.

## نمونه اجرا

نمونه ای از اجرای این کد را میبینید.



The screenshot shows an IDE with two tabs: `QLearningDatacenterBroker.java` and `FaultToleranceManager.java`. The `FaultToleranceManager.java` tab is active, showing the following code:

```
public static List<Cloudlet> loadCheckpointData() {
    try {
        FileInputStream fis = new FileInputStream(CHECKPOINT_FILE_PATH);
        ObjectInputStream ois = new ObjectInputStream(fis);
        return (List<Cloudlet>) ois.readObject();
    } catch (IOException | ClassNotFoundException e) {
        e.printStackTrace();
    }
    return null;
}

public static void saveCheckpointData(List<Cloudlet> list) {
    try {
        FileOutputStream fos = new FileOutputStream(CHECKPOINT_FILE_PATH);
        ObjectOutputStream oos = new ObjectOutputStream(fos);
    }
}
```

The console output shows the following messages:

```
Datacenter_181 is shutting down...
Datacenter_182 is shutting down...
Datacenter_183 is shutting down...
Datacenter_184 is shutting down...
Datacenter_185 is shutting down...
Datacenter_186 is shutting down...
Datacenter_187 is shutting down...
Datacenter_188 is shutting down...
Datacenter_189 is shutting down...
Datacenter_190 is shutting down...
Datacenter_191 is shutting down...
Datacenter_192 is shutting down...
Datacenter_193 is shutting down...
Datacenter_194 is shutting down...
Datacenter_195 is shutting down...
Datacenter_196 is shutting down...
Datacenter_197 is shutting down...
Datacenter_198 is shutting down...
Datacenter_199 is shutting down...
Broker_0 is shutting down...
Simulation completed.
Simulation completed.
QLearning.QLearningScheduler finished!
Data saved to data/200_1000/qlearning_data.csv

Process finished with exit code 0
```

در سیستم محلی ممن حدود ۳۵ ثانیه شبیه سازی و اجرای کامل این کد زمان برد. (رم ۱۶ - Windows ۱۰).

برای بهبود زمان اجرا میتواند log ها را پرینت نکرد.

## منابع

می توانید به سورس کد این پروژه از طریق [این لینک](#) دسترسی داشته باشید.

البته با توجه به خصوصی بودن این رپازیتوری، سورس کد مربوطه در فایللی کنار آن بارگذاری شده است.

با تشکر.

## تغییرات فاز ۲

در این فاز، الگوریتم Double QLearning اضافه شده است که به جلی یک QTable. ۲ عدد QTable داریم و در تابع `getBestAction`، مجموع ارزش هلی این ۲ را به عنوان ارزش کلی حساب میکنیم.

همچنین در هنگام به روزرسانی QTable نیز به احتمال  $\frac{1}{2}$  از جدول ۱ پیروی میکنیم و به احتمال  $\frac{1}{2}$  از جدول ۲.

همچنین با اضافه کردن این خطوط، میتوانیم CPU Utilization را نیز در خروجی نمایش دهیم.

```
String cost = dft.format( number: cloudlet.getCostPerSec() * cloudlet.getActualCPUTime());
int dcId = cloudlet.getVmId() % Constants.NO_OF_DATA_CENTERS;
String cpuUtilization = dft.format( number: cloudlet.getActualCPUTime() / execMatrix[i][dcId]);

String[] row = {cloudletId, status, dataCenterId, vmId, time, startTime, finishTime, waitingTime,
                completionTime, cost, cpuUtilization};
writer.writeNext(row);
```

تنها موردی که نیاز است همچنان پیاده سازی شود، بحث (dynamic priority queue based on deadline) است که در صدد پیاده سازی آن هستیم.