UNIVERSITÉ TOULOUSE III PAUL SABATIER

Simulation of a Task Scheduler (M1 CSA - Report)

Author(s): Florian Legendre, Niels Boulanger

Contents

Part 1

Monoserver Scheduling

Principles for the implementation

For all algorithms applied to single servers presented here the initial situation will be the same as illustrated here:

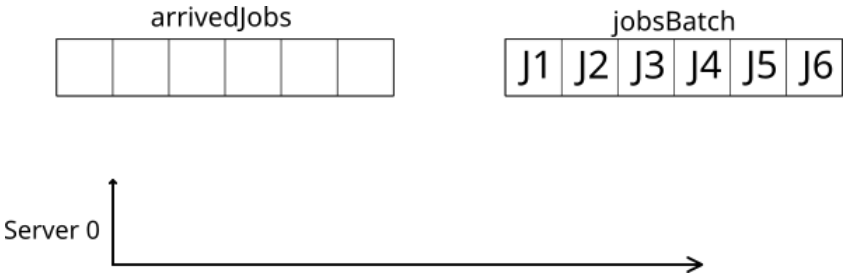

Figure 1: Initial situation of a monoserver scheduling

We start with a jobs batch of loaded jobs from the test configuration file. A jobs batch is a list of jobs (or in our code a specialized class containing such a list). This jobs batch is sorted by order of arrivals. We initialize our scheduling algorithm by fetching the soonest jobs to arrive and adding them to the arrivedJobs list. We fetch the soonest jobs as we can't now for sure when is the first arrival date and how many jobs arrive at this same arrival dates.

Once the arrivedJobs list is initialized by fetching the soonest jobs to arrive we execute scheduling steps and we loop on these scheduling steps as long as the jobs batch and the arrivedJobs list aren't empty. A special case may happen at some point where the jobs batch isn't empty but the arrivedJobs is empty and the server isn't running any job. In this case we reinitialize our scheduling by fetching the soonest jobs to arrive again.

This common logic to all scheduling is what led us to implement this common loop used by all scheduling algorithms even in the next parts of this report:

```
1 protected void run(){
2     while( !(jobsB.isEmpty() && arrivedJ.isEmpty() && serversM.areAllServersIdle
      ()) ){
3         if(serversM.areAllServersIdle() && arrivedJ.isEmpty()) arrivedJ.addAll(
      jobsB.getSoonestJobs());
4         runScheduleStep();
5     }
6 }
```

Listing 1: Main scheduling loop in Scheduler abstract class (cf. run() method)

Another major principle of our scheduling algorithms used throughout all the exercises is the event-based system of scheduling steps. Instead of using time steps of 1 and checking at each time step the environment we use event steps. There are only two kinds of events we have to deal with so it is still easily manageable:

1. A job has just finished
2. A job has just arrived

Inside each schedule step we will have to compute the next event date based on these two possible events which is why we have the following method inside the Scheduler abstract class from which all scheduling algorithms inherit:

```
1 protected double getNextEventDate(){
2     double nextArrivalDate = jobsB.getNextArrivalDate();
3     double nextJobToFinishDate = schedule.currentDate + serversM.
      getNextServerToFinish().getDuration();
4
5     if(nextArrivalDate == -1) return nextJobToFinishDate;
6     else return Math.min(nextArrivalDate, nextJobToFinishDate);
7 }
```

Listing 2: Method used to compute the next event fate

When an event occurs we will have to decide if an entry needs to be computed in the schedule[1] and by how much we have to decrement the currently running job's amount of units of work. We then fetch the jobs that have arrived in the meantime and we assign the newly arrived tasks to the server according to the specific algorithm assigning policy. Finally, we start another scheduling step which will "stop" when another another event occurs.

As you may notice even if all schedule steps have a common structure to manage the arrivedJobs list, the jobsBatch and also the current date of the schedule being computed, the further we get inside the schedule step the more it gets specific to the scheduling algorithm we choose. This is why the method *runScheduleStep()* is abstract and is left to be redefined in each algorithm of this labwork.

Finally, we use a few helper classes such as the ServerManager class that provide functions to manage and observe the servers state through all the scheduling process. Which will lead us to use a few contractions:

- serversM → serversManager as presented in section 1.1
- arrivedJ → arrivedJobs
- jobsB → jobsBatch

[1] For example if a job has just finished or if it has been preempted

FIFO

1.2.1 Implementation

As the FIFO has been adapted to run on multiple servers, the *decrementAll()* and *assignArrivals()* methods take care of outputting entries in the Schedule object contained in the scheduler. One possible improvement on this labwork could be to further generalize the structure of a schedule step for all algorithms. Indeed, the illustrated code source will be very similar for all scheduling algorithms presented here.

```
1  public void runScheduleStep() {
2      if(serversM.areAllServersIdle() && !arrivedJ.isEmpty()){
3          schedule.currentDate = arrivedJ.getFirst().getArrivalDate();
4          serversM.initServers();
5      }
6
7      //We compute next event date:
8      double nextEventDate = getNextEventDate();
9      double unitsOfWorkDone = nextEventDate - schedule.currentDate;
10     schedule.currentDate += unitsOfWorkDone;
11
12     //We decrement and deal with finished jobs:
13     serversM.decrementAll(unitsOfWorkDone);
14
15     //We deal with new arrivals:
16     arrivedJ.addAll(jobsB.getArrivedJobs(nextEventDate));
17     assignArrivals();
18 }
```

Listing 3: Source code of a FIFO schedule step

1.2.2 Scheduling results

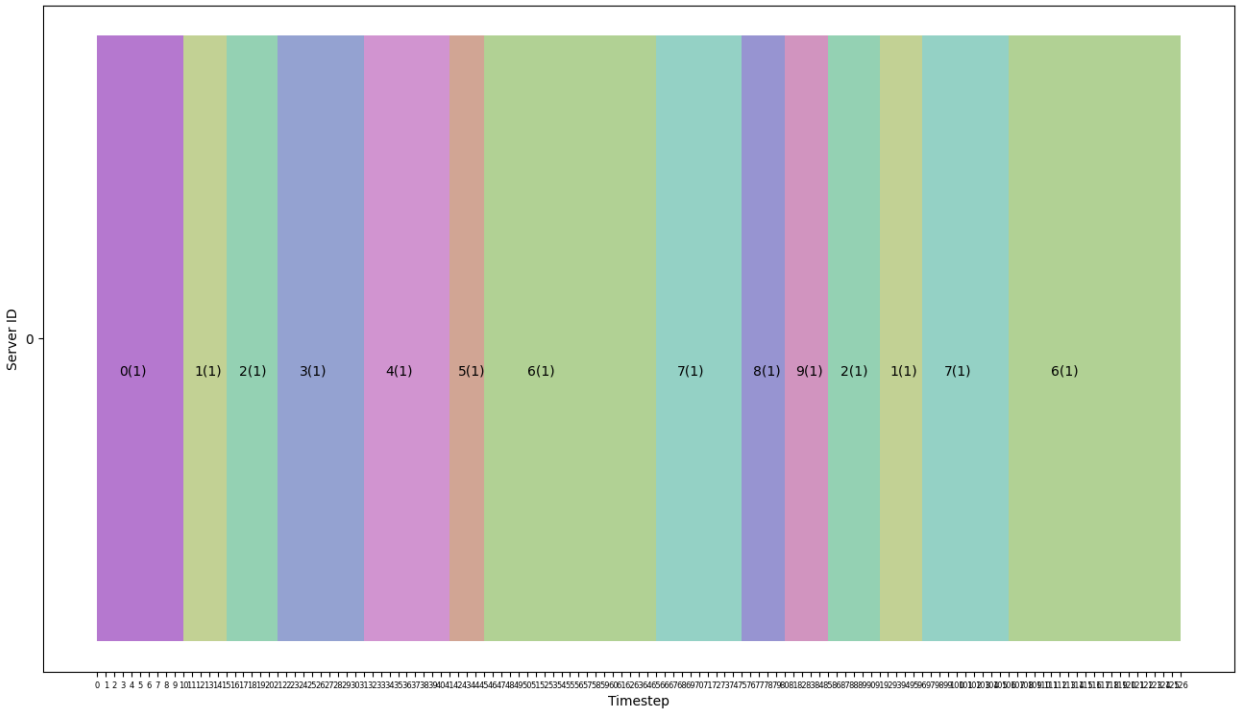

Figure 2: Output schedule produced by the FIFO scheduling algorithm on one server

Which corresponds to following file (<jobID, serverID, startDate, endDate, frequency>):

```
0 0 0.0 10.0 1.0
1 0 10.0 15.0 1.0
2 0 15.0 21.0 1.0
3 0 21.0 31.0 1.0
4 0 31.0 41.0 1.0
5 0 41.0 45.0 1.0
6 0 45.0 65.0 1.0
7 0 65.0 75.0 1.0
8 0 75.0 80.0 1.0
9 0 80.0 85.0 1.0
2 0 85.0 91.0 1.0
1 0 91.0 96.0 1.0
7 0 96.0 106.0 1.0
6 0 106.0 126.0 1.0
```

Listing 4: Detail of the scheduling result

1.2.3 Metrics

```
################## SCHEDULE METRICS ##################

>> Scheduling Metrics:
- Total Makespan: 126.0
- Nb Deadline Misses: 11
- Max Tardiness: 69.0
- Average Tardiness: 36.63636363636363
- Late Jobs:
[Job: J6{6.0a/0.0u/30.0rd/36.0ad/100.0p} | Tardiness: 29.0 ]
[Job: J2{3.0a/0.0u/10.0rd/13.0ad/10.0p} | Tardiness: 8.0 ]
[Job: J2{13.0a/0.0u/10.0rd/23.0ad/10.0p} | Tardiness: 68.0 ]
[Job: J8{11.0a/0.0u/5.0rd/16.0ad/0.0p} | Tardiness: 64.0 ]
[Job: J4{5.0a/0.0u/30.0rd/35.0ad/0.0p} | Tardiness: 6.0 ]
[Job: J7{10.0a/0.0u/25.0rd/35.0ad/50.0p} | Tardiness: 40.0 ]
[Job: J7{60.0a/0.0u/25.0rd/85.0ad/50.0p} | Tardiness: 21.0 ]
[Job: J1{0.0a/0.0u/10.0rd/10.0ad/20.0p} | Tardiness: 5.0 ]
[Job: J5{6.0a/0.0u/12.0rd/18.0ad/0.0p} | Tardiness: 27.0 ]
[Job: J9{11.0a/0.0u/5.0rd/16.0ad/0.0p} | Tardiness: 69.0 ]
[Job: J1{20.0a/0.0u/10.0rd/30.0ad/20.0p} | Tardiness: 66.0 ]

>> Servers Metrics:
- Servers work load:
Server #0 : 126.0

>> Energy Metrics:
- Total Consumption: 2799.999999999995
- Max Consumption: 22.22222222222222
- Average Consumption: 2799.999999999995
- Consumption per Server:
Server #0 : 2799.999999999995

######################################################
```

Listing 5: Metrics for FIFO on a single server

Round Robin

1.3.1 Implementation

Contrary to the FIFO algorithm which has been adapted to run on multiple servers this algorithm only works on a single server. One improvement on this labwork could be to develop a multiserver version of round robin.

```
1     public void runScheduleStep() {
2         Job job = arrivedJ.removeFirst();
3
4         //We compute next event date:
5         double start = ScheduleEntry.computeStart(schedule, serversM.getServers()
      .getFirst(), job);
6         double end = ScheduleEntry.computeEnd(job, start, QUANTUM);
7         schedule.currentDate = end;
8
9         //We decrement and deal with the finished job:
10        job.decrement(QUANTUM);
11        ScheduleEntry newEntry = new ScheduleEntry(job, serversM.getServers().
      getFirst(), start, end,
12        serversM.getServers().getFirst().getCurrFreq());
13        schedule.add(newEntry);
14
15        //We deal with new arrivals:
16        arrivedJ.addAll(jobsB.getArrivedJobs(end));
17        if(!job.isWorkDone()) arrivedJ.add(job);
18    }
```

Listing 6: Source code of a Round Robin **on a single server** schedule step

1.3.2 Scheduling results

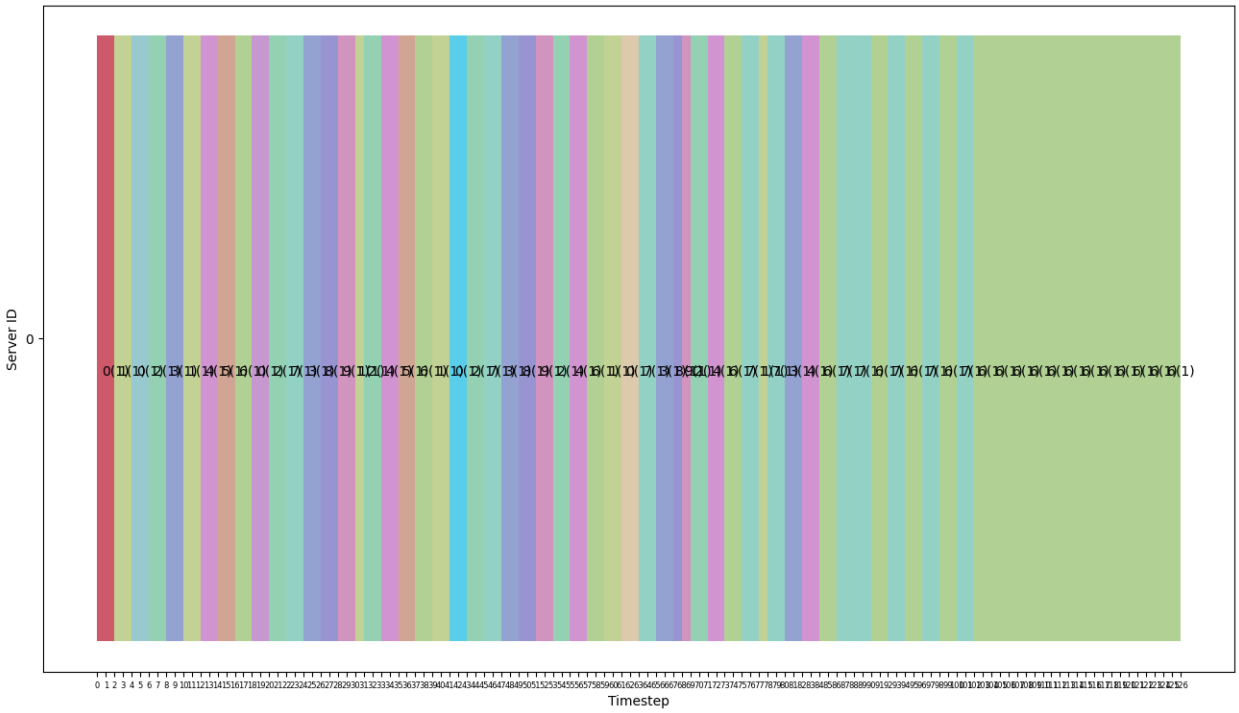

Figure 3: Output schedule produced by the Round Robin scheduling algorithm on one server

Which corresponds to following file (<jobID, serverID, startDate, endDate, frequency>):

```
0 0 0.0 2.0 1.0
1 0 2.0 4.0 1.0
0 0 4.0 6.0 1.0
2 0 6.0 8.0 1.0
3 0 8.0 10.0 1.0
1 0 10.0 12.0 1.0
4 0 12.0 14.0 1.0
5 0 14.0 16.0 1.0
6 0 16.0 18.0 1.0
0 0 18.0 20.0 1.0
2 0 20.0 22.0 1.0
7 0 22.0 24.0 1.0
3 0 24.0 26.0 1.0
8 0 26.0 28.0 1.0
9 0 28.0 30.0 1.0
1 0 30.0 31.0 1.0
2 0 31.0 33.0 1.0
4 0 33.0 35.0 1.0
5 0 35.0 37.0 1.0
6 0 37.0 39.0 1.0
1 0 39.0 41.0 1.0
0 0 41.0 43.0 1.0
2 0 43.0 45.0 1.0
7 0 45.0 47.0 1.0
3 0 47.0 49.0 1.0
8 0 49.0 51.0 1.0
```

```
9 0 51.0 53.0 1.0
2 0 53.0 55.0 1.0
4 0 55.0 57.0 1.0
6 0 57.0 59.0 1.0
1 0 59.0 61.0 1.0
0 0 61.0 63.0 1.0
7 0 63.0 65.0 1.0
3 0 65.0 67.0 1.0
8 0 67.0 68.0 1.0
9 0 68.0 69.0 1.0
2 0 69.0 71.0 1.0
4 0 71.0 73.0 1.0
6 0 73.0 75.0 1.0
7 0 75.0 77.0 1.0
1 0 77.0 78.0 1.0
7 0 78.0 80.0 1.0
3 0 80.0 82.0 1.0
4 0 82.0 84.0 1.0
6 0 84.0 86.0 1.0
7 0 86.0 88.0 1.0
7 0 88.0 90.0 1.0
6 0 90.0 92.0 1.0
7 0 92.0 94.0 1.0
6 0 94.0 96.0 1.0
7 0 96.0 98.0 1.0
6 0 98.0 100.0 1.0
7 0 100.0 102.0 1.0
6 0 102.0 104.0 1.0
6 0 104.0 106.0 1.0
6 0 106.0 108.0 1.0
6 0 108.0 110.0 1.0
6 0 110.0 112.0 1.0
6 0 112.0 114.0 1.0
6 0 114.0 116.0 1.0
6 0 116.0 118.0 1.0
6 0 118.0 120.0 1.0
6 0 120.0 122.0 1.0
6 0 122.0 124.0 1.0
6 0 124.0 126.0 1.0
```

Listing 7: Detail of the scheduling result

1.3.3 Metrics

```
################## SCHEDULE METRICS ##################

>> Scheduling Metrics:
- Total Makespan: 126.0
- Nb Deadline Misses: 13
- Max Tardiness: 70.0
- Average Tardiness: 43.07692307692308
- Late Jobs:
[Job: J0{0.0a/0.0u/15.0rd/15.0ad/0.0p} | Tardiness: 48.0 ]
[Job: J9{11.0a/0.0u/5.0rd/16.0ad/0.0p} | Tardiness: 53.0 ]
[Job: J7{10.0a/0.0u/25.0rd/35.0ad/50.0p} | Tardiness: 55.0 ]
[Job: J5{6.0a/0.0u/12.0rd/18.0ad/0.0p} | Tardiness: 19.0 ]
[Job: J8{11.0a/0.0u/5.0rd/16.0ad/0.0p} | Tardiness: 52.0 ]
[Job: J1{0.0a/0.0u/10.0rd/10.0ad/20.0p} | Tardiness: 21.0 ]
[Job: J3{4.0a/0.0u/30.0rd/34.0ad/0.0p} | Tardiness: 48.0 ]
[Job: J4{5.0a/0.0u/30.0rd/35.0ad/0.0p} | Tardiness: 49.0 ]
[Job: J2{13.0a/0.0u/10.0rd/23.0ad/10.0p} | Tardiness: 48.0 ]
[Job: J1{20.0a/0.0u/10.0rd/30.0ad/20.0p} | Tardiness: 48.0 ]
[Job: J2{3.0a/0.0u/10.0rd/13.0ad/10.0p} | Tardiness: 32.0 ]
[Job: J6{6.0a/0.0u/30.0rd/36.0ad/100.0p} | Tardiness: 70.0 ]
[Job: J7{60.0a/0.0u/25.0rd/85.0ad/50.0p} | Tardiness: 17.0 ]

>> Servers Metrics:
- Servers work load:
Server #0 : 126.0

>> Energy Metrics:
- Total Consumption: 2799.999999999995
- Max Consumption: 22.22222222222222
- Average Consumption: 2799.999999999995
- Consumption per Server:
Server #0 : 2799.999999999995

######################################################
```

Listing 8: Metrics for Round Robin on a single server

EDF

1.4.1 Implementation

Contrary to FIFO and round robin we notice that the arrived jobs list is now sorted based on a comparison key between jobs. For EDF this comparison key corresponds to the absolute deadline[2]. Also a jobs comparison predicate is used to allow us to compare two jobs (and not only a whole list) based on a criteria (which is still the absolute deadline here).

```
1   protected void runScheduleStep() {
2       arrivedJ.sort(JOBS_COMPARISON_KEY);
3       if(serversM.areAllServersIdle() && !arrivedJ.isEmpty()){
4           schedule.currentDate = arrivedJ.getFirst().getArrivalDate();
5           serversM.initServers();
6       }
7
8       //We compute next event date:
9       double nextEventDate = getNextEventDate();
10      double unitsOfWorkDone = nextEventDate - schedule.currentDate;
11      schedule.currentDate += unitsOfWorkDone;
12
13      //We decrement and deal with finished jobs:
14      serversM.decrementAll(unitsOfWorkDone);
15
16      //We deal with new arrivals:
17      arrivedJ.addAll(jobsB.getArrivedJobs(nextEventDate));
18      arrivedJ.sort(JOBS_COMPARISON_KEY);
19      assignArrivals(JOBS_COMPARISON_KEY, JOBS_COMPARISON_PREDICATE);
20  }
```

Listing 9: Source code of an EDF schedule step

[2]arrival date + relative deadline of the job

1.4.2 Scheduling results

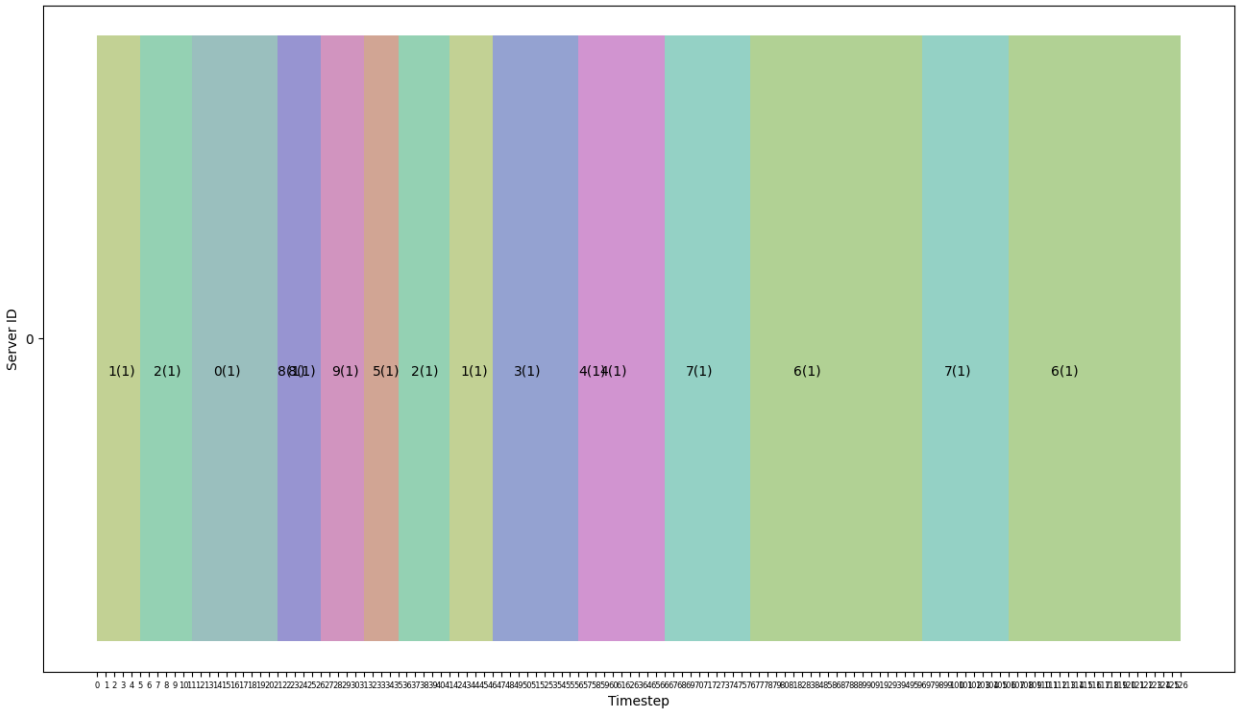

Figure 4: Output schedule produced by the EDF scheduling algorithm on one server

Which corresponds to following file (<jobID, serverID, startDate, endDate, frequency>):

```
1 0 0.0 5.0 1.0
2 0 5.0 11.0 1.0
0 0 11.0 21.0 1.0
8 0 21.0 21.0 1.0
8 0 21.0 26.0 1.0
9 0 26.0 31.0 1.0
5 0 31.0 35.0 1.0
2 0 35.0 41.0 1.0
1 0 41.0 46.0 1.0
3 0 46.0 56.0 1.0
4 0 56.0 56.0 1.0
4 0 56.0 66.0 1.0
7 0 66.0 76.0 1.0
6 0 76.0 96.0 1.0
7 0 96.0 106.0 1.0
6 0 106.0 126.0 1.0
```

Listing 10: Detail of the scheduling result

1.4.3 Metrics

```
################## SCHEDULE METRICS ##################

>> Scheduling Metrics:
- Total Makespan: 126.0
- Nb Deadline Misses: 11
- Max Tardiness: 60.0
- Average Tardiness: 23.363636363636363
- Late Jobs:
[Job: J2{13.0a/0.0u/10.0rd/23.0ad/10.0p} | Tardiness: 18.0 ]
[Job: J6{6.0a/0.0u/30.0rd/36.0ad/100.0p} | Tardiness: 60.0 ]
[Job: J3{4.0a/0.0u/30.0rd/34.0ad/0.0p} | Tardiness: 22.0 ]
[Job: J5{6.0a/0.0u/12.0rd/18.0ad/0.0p} | Tardiness: 17.0 ]
[Job: J0{0.0a/0.0u/15.0rd/15.0ad/0.0p} | Tardiness: 6.0 ]
[Job: J9{11.0a/0.0u/5.0rd/16.0ad/0.0p} | Tardiness: 15.0 ]
[Job: J7{60.0a/0.0u/25.0rd/85.0ad/50.0p} | Tardiness: 21.0 ]
[Job: J4{5.0a/0.0u/30.0rd/35.0ad/0.0p} | Tardiness: 31.0 ]
[Job: J8{11.0a/0.0u/5.0rd/16.0ad/0.0p} | Tardiness: 10.0 ]
[Job: J7{10.0a/0.0u/25.0rd/35.0ad/50.0p} | Tardiness: 41.0 ]
[Job: J1{20.0a/0.0u/10.0rd/30.0ad/20.0p} | Tardiness: 16.0 ]

>> Servers Metrics:
- Servers work load:
Server #0 : 126.0

>> Energy Metrics:
- Total Consumption: 2799.999999999995
- Max Consumption: 22.22222222222222
- Average Consumption: 2799.999999999995
- Consumption per Server:
Server #0 : 2799.999999999995

######################################################
```

Listing 11: Metrics for EDF on a single server

1.5.1 Implementation

As mentioned in the previous subsection 1.4.1 on the EDF implementation the algorithm is basically the same. Only the comparison keys and predicates change. Indeed, in EDF we have the following comparison keys/predicates:

```
1  private static final Comparator<Job> JOBS_COMPARISON_KEY = Comparator.
   comparingDouble(Job::getADeadline);
2  private static final BiPredicate<Job, Job> JOBS_COMPARISON_PREDICATE = (Job
   j1, Job j2) -> j1.getADeadline() < j2.getADeadline();
```

Listing 12: Comparison key and predicate of EDF

While on RMS we have the more complex ones here:

```
1  private static final Comparator<Job> JOBS_COMPARISON_KEY = (j1, j2) -> {
2      if (j1.getPeriod() == j2.getPeriod())
3      return Double.compare(j1.getADeadline(), j2.getADeadline());
4      else
5      return Double.compare(j1.getPeriod(), j2.getPeriod());
6  };
7  private static final BiPredicate<Job, Job> JOBS_COMPARISON_PREDICATE =  (j1,
   j2) -> {
8      if (j1.getPeriod() == j2.getPeriod())
9      return j1.getADeadline() <= j2.getADeadline();
10     else
11     return j1.getPeriod() <= j2.getPeriod();
12 };
```

Listing 13: Comparison key and predicate of RMS

This comparison key/predicate for RMS allows us to first sort/compare the tasks using their period (smallest periods being the most important) then, if the jobs are without periods (period = 0), to sort/compare them according to their absolute deadlines (just like with EDF).

1.5.2 Scheduling results

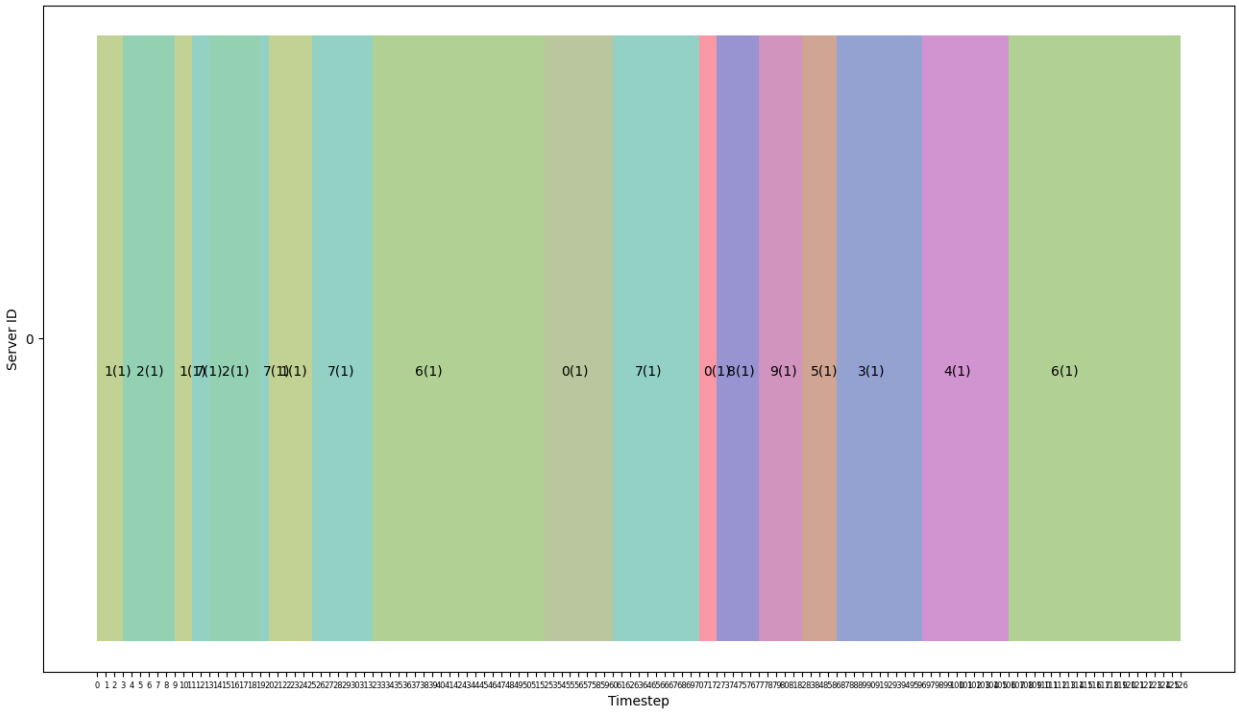

Figure 5: Output schedule produced by the RMS scheduling algorithm on one server

Which corresponds to following file (<jobID, serverID, startDate, endDate, frequency>):

```
1 0 0.0 3.0 1.0
2 0 3.0 9.0 1.0
1 0 9.0 11.0 1.0
7 0 11.0 13.0 1.0
2 0 13.0 19.0 1.0
7 0 19.0 20.0 1.0
1 0 20.0 25.0 1.0
7 0 25.0 32.0 1.0
6 0 32.0 52.0 1.0
0 0 52.0 60.0 1.0
7 0 60.0 70.0 1.0
0 0 70.0 72.0 1.0
8 0 72.0 77.0 1.0
9 0 77.0 82.0 1.0
5 0 82.0 86.0 1.0
3 0 86.0 96.0 1.0
4 0 96.0 106.0 1.0
6 0 106.0 126.0 1.0
```

Listing 14: Detail of the scheduling result

1.5.3 Metrics

```
################## SCHEDULE METRICS ##################

>> Scheduling Metrics:
- Total Makespan: 126.0
- Nb Deadline Misses: 8
- Max Tardiness: 71.0
- Average Tardiness: 50.25
- Late Jobs:
[Job: J4{5.0a/0.0u/30.0rd/35.0ad/0.0p} | Tardiness: 71.0 ]
[Job: J1{0.0a/0.0u/10.0rd/10.0ad/20.0p} | Tardiness: 1.0 ]
[Job: J6{6.0a/0.0u/30.0rd/36.0ad/100.0p} | Tardiness: 16.0 ]
[Job: J8{11.0a/0.0u/5.0rd/16.0ad/0.0p} | Tardiness: 61.0 ]
[Job: J5{6.0a/0.0u/12.0rd/18.0ad/0.0p} | Tardiness: 68.0 ]
[Job: J9{11.0a/0.0u/5.0rd/16.0ad/0.0p} | Tardiness: 66.0 ]
[Job: J0{0.0a/0.0u/15.0rd/15.0ad/0.0p} | Tardiness: 57.0 ]
[Job: J3{4.0a/0.0u/30.0rd/34.0ad/0.0p} | Tardiness: 62.0 ]

>> Servers Metrics:
- Servers work load:
Server #0 : 126.0

>> Energy Metrics:
- Total Consumption: 2799.999999999995
- Max Consumption: 22.22222222222222
- Average Consumption: 2799.999999999995
- Consumption per Server:
Server #0 : 2799.999999999995

######################################################
```

Listing 15: Metrics for RMS on a single server

Comparison Table

Algorithm	Pros	Cons	Possible Uses
FIFO	• Easy to implement and debug • Corresponds to the expected behavior of many applications (e.g. printers, chains of tasks such as piped commands, etc.)	• Risks of starvation if a task is blocked • Risks of heavy tardiness if a task takes an exceptionally long amount of time to complete	• When the application requires tasks to execute in a consecutive order (could be made safer with a timeout mechanic)
Round Robin	• Prevents starvation • Still rather easy to implement and debug	• Tends to even more increase the number of deadline misses and average tardiness of tasks if the quantum is short	• With a reasonable quantum and if a strict sequence in the order of treated tasks isn't needed as in FIFO, it can be a fair way to run tasks while being starvation-free.
EDF	• Lowest tardiness of tasks (but not necessarily lowest number of deadline misses)	• A bit more complex to implement and maintain • Not starvation free as if no new arrival with higher priority comes then priorities are fixed and a blocked running task can block all the other tasks	• Useful if we want highly responsive applications
RMS	• Lowest number of deadline misses	• A bit more complex to implement and maintain (but once you have implemented EDF it's rather straightforward and reciprocally) • Not starvation free for the same reasons than with EDF	• Probably the best algorithm for hard real-time operating systems where deadline misses isn't an option

Part 2

Multiserver Scheduling

Principles for the implementation

In the single server version the *arrivedJobs* list could be used as an *assignedJobs* list. But with multiple servers it became difficult to keep track of which server had started to run each task in the event of a preemption (a task that starts on a server must end on the same server). So we added to the *Server* class an *assignedJobs* list so that each server knows which jobs it has the responsibility of.

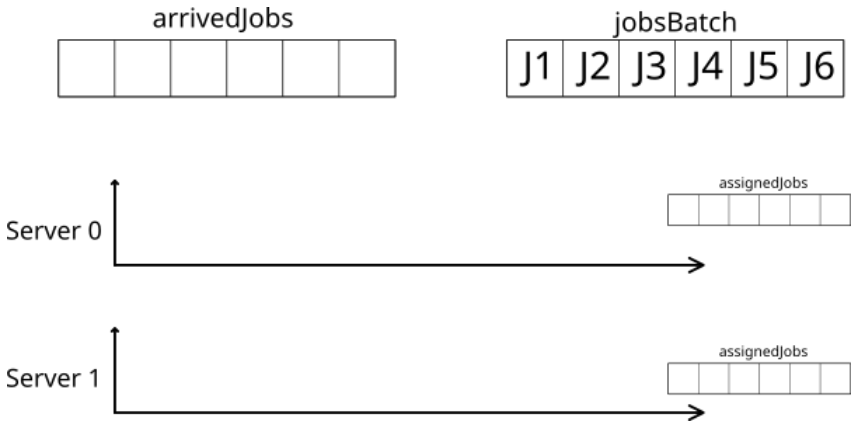

Figure 6: Initial situation of a multiserver scheduling

In this configuration, jobs from the jobs batch are still by default assigned to the *arrivedJobs* list whenever the current date in the schedule allows it. What is new is that:

- We have to check across all servers for the next event date
- We have to decrement all jobs across all servers
- We have to choose on which server to assign newly arrived tasks

We provide the example of the arrival of a job 4 in the following illustration:

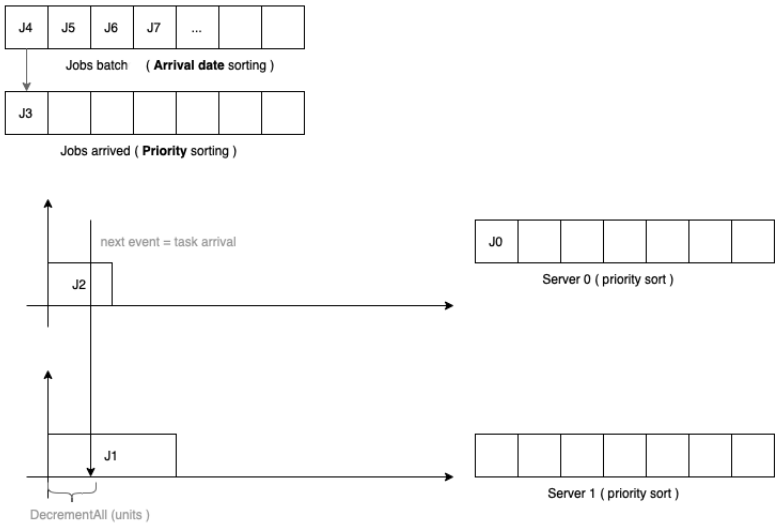

Figure 7: Illustration of the arrival of a job 4 computed across all servers with the problem of its assignment

The method *assignArrivals()* manages the arrival assignments based on the specifications of our algorithms. For example, in EDF, arrived tasks are assigned to a server if:

1. The server is idle
2. The server has a task with a lower priority (ie. lower absolute deadline)

If none of these conditions are verified then the task waits in the *arrivedJobs* list. Servers that are idle will fetch the most critical task themselves if they are idle (cf. subsection 1.4.1 for the corresponding source code).

This arrival specification varying from algorithm to algorithm they are redefined in their abstract classes *SchedulerQuantum* and *SchedulerPriority* which consistute families of schedulers[3]. We provide here an example of an arrival assignment implementation used by the *SchedulerPriority* family:

```
1   protected void assignArrivals(Comparator<Job> jobsComparKey, BiPredicate<Job,
    Job> jobsComparPredicate){
2       boolean isAssignable = true;
3       boolean isAssigned;
4       Iterator<Job> jobIterator = arrivedJ.iterator();
5
6       while(jobIterator.hasNext() && isAssignable){
7           Job j = jobIterator.next();
8           isAssigned = serversM.assignToIdle(j);
9
10          if(isAssigned) jobIterator.remove();
11          else{
12              Server servP = getPreemptable(j, jobsComparPredicate);
13              if(servP == null) isAssignable = false;
14              else{
15                  double start = ScheduleEntry.computeStart(schedule, servP,
    servP.getRunningJob());
16                  schedule.add(new ScheduleEntry(servP.getRunningJob(), servP,
    start, schedule.currentDate, servP.getCurrFreq()));
17                  servP.getAssignedJobs().add(j);
18                  servP.getAssignedJobs().sort(jobsComparKey);
19                  jobIterator.remove();
20              }
21          }
22      }
23  }
```

Listing 16: Source code of the *assignArrivals()* used by the *SchedulerPriority* family of Schedulers

[3]FIFO and Round Robin corresponding to the *SchedulerQuantum* family and EDF/RMS to the *SchedulerPriority* family

EDF

2.8.1 Implementation

The algorithm used is the same as presented in subsection 1.4.1 of the previous part of this report.

2.8.2 Scheduling results

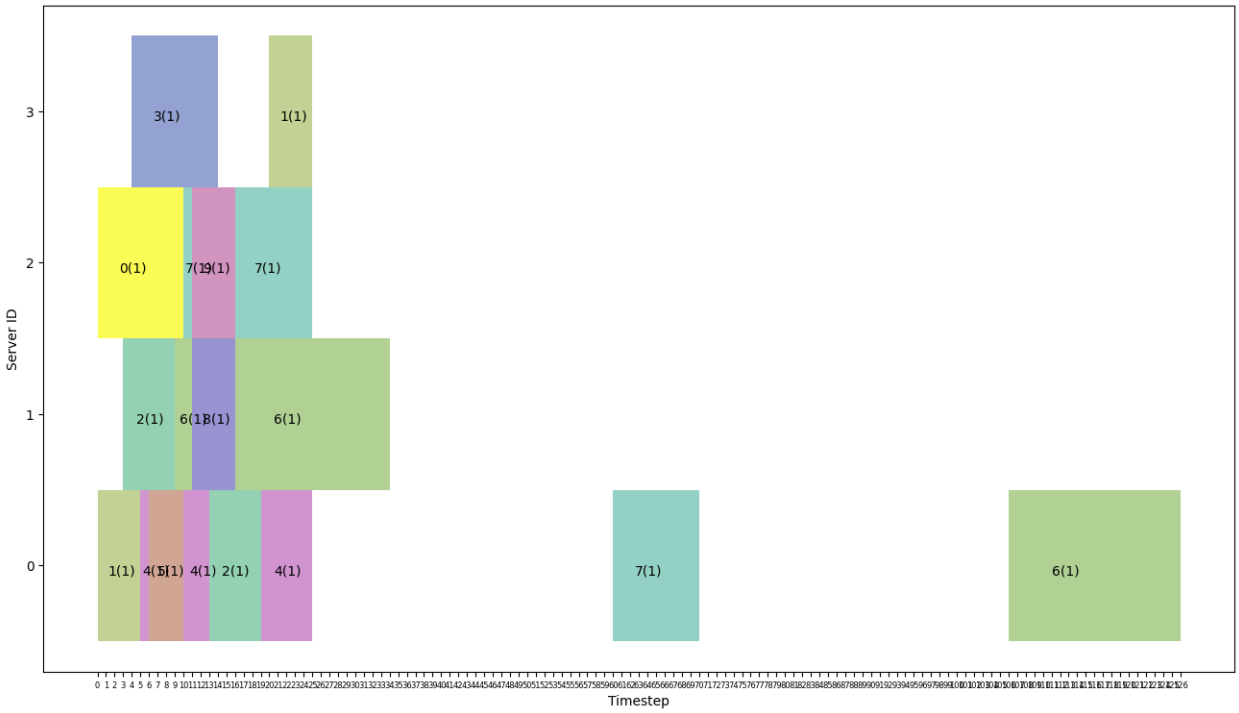

Figure 8: Output schedule produced by the EDF scheduling algorithm on multiple servers

Which corresponds to following file (<jobID, serverID, startDate, endDate, frequency>):

```
1 0 0.0 5.0 1.0
4 0 5.0 6.0 1.0
2 1 3.0 9.0 1.0
5 0 6.0 10.0 1.0
0 2 0.0 10.0 1.0
6 1 9.0 11.0 1.0
7 2 10.0 11.0 1.0
4 0 10.0 13.0 1.0
3 3 4.0 14.0 1.0
8 1 11.0 16.0 1.0
9 2 11.0 16.0 1.0
2 0 13.0 19.0 1.0
4 0 19.0 25.0 1.0
7 2 16.0 25.0 1.0
1 3 20.0 25.0 1.0
6 1 16.0 34.0 1.0
7 0 60.0 70.0 1.0
6 0 106.0 126.0 1.0
```

Listing 17: Detail of the scheduling result

2.8.3 Metrics

```
################## SCHEDULE METRICS ##################

>> Scheduling Metrics:
- Total Makespan: 126.0
- Nb Deadline Misses: 0
- Max Tardiness: 0.0
- Average Tardiness: 0.0
- Late Jobs:

>> Servers Metrics:
- Servers work load:
Server #0 : 55.0
Server #1 : 31.0
Server #2 : 25.0
Server #3 : 15.0

>> Energy Metrics:
- Total Consumption: 4077.777777777775
- Max Consumption: 144.44444444444446
- Average Consumption: 1019.4444444444438
- Consumption per Server:
Server #0 : 1222.2222222222208
Server #1 : 1550.0
Server #2 : 555.5555555555555
Server #3 : 750.0

######################################################
```

Listing 18: Metrics for EDF on multiple servers

RMS

2.9.1 Implementation

The algorithm used is the same as presented in subsection 1.5.1 of the previous part of this report.

2.9.2 Scheduling results

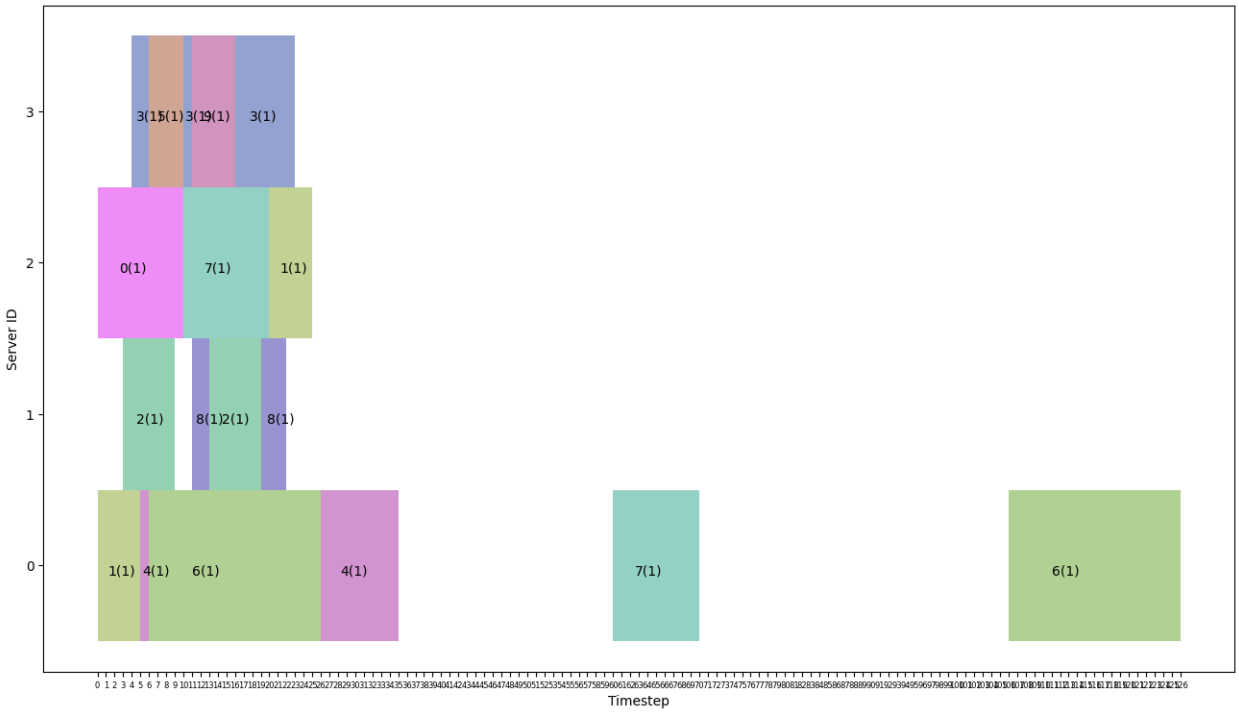

Figure 9: Output schedule produced by the RMS scheduling algorithm on multiple servers

Which corresponds to following file (<jobID, serverID, startDate, endDate, frequency>):

```
1 0 0.0 5.0 1.0
4 0 5.0 6.0 1.0
3 3 4.0 6.0 1.0
2 1 3.0 9.0 1.0
0 2 0.0 10.0 1.0
5 3 6.0 10.0 1.0
3 3 10.0 11.0 1.0
8 1 11.0 13.0 1.0
9 3 11.0 16.0 1.0
2 1 13.0 19.0 1.0
7 2 10.0 20.0 1.0
8 1 19.0 22.0 1.0
3 3 16.0 23.0 1.0
1 2 20.0 25.0 1.0
6 0 6.0 26.0 1.0
4 0 26.0 35.0 1.0
7 0 60.0 70.0 1.0
6 0 106.0 126.0 1.0
```

Listing 19: Detail of the scheduling result

2.9.3 Metrics

```
################## SCHEDULE METRICS ##################

>> Scheduling Metrics:
- Total Makespan: 126.0
- Nb Deadline Misses: 1
- Max Tardiness: 6.0
- Average Tardiness: 6.0
- Late Jobs:
[Job: J8{11.0a/0.0u/5.0rd/16.0ad/0.0p} | Tardiness: 6.0 ]

>> Servers Metrics:
- Servers work load:
Server #0 : 65.0
Server #1 : 17.0
Server #2 : 25.0
Server #3 : 19.0

>> Energy Metrics:
- Total Consumption: 3799.999999999997
- Max Consumption: 144.44444444444446
- Average Consumption: 949.9999999999992
- Consumption per Server:
Server #0 : 1444.4444444444425
Server #1 : 850.0
Server #2 : 555.5555555555555
Server #3 : 950.0

#######################################################
```

Listing 20: Metrics for RMS on multiple servers

Comparison Table

Algorithm	Pros	Cons	Possible Uses
EDF	• With several servers (or cores): lowest number of deadline misses and lowest average tardiness • Best workload share among servers	• What if the deadline is superior to the period of the task? • Rather hard to implement and maintain on several servers (lot of lists sorting)	• May be the best algorithm for hard real time OS if the deadline is inferior to the period of the task
RMS	• Very low number of deadline misses and average tardiness	• Rather hard to implement for the same reasons than with EDF • Not the best algorithm for hard real time OS if the deadline of the tasks is inferior to their periods?	• May still be the best algorithm for hard real time OS if tasks' deadlines are equal or superior to their periods

Part 3

Multiserver Energy-aware Scheduling

Principles for the implementation

For this implementation of these energy aware versions of the scheduling algorithms we established a couple of hypothesis:

- Energy aware means that we want the smallest possible consumption (and not the highest one given a maximum power consumption constraint)
- All servers can run simultaneously if they all use their minimal frequencies and if they can't the problem is equivalent to having less servers all running on their minimal frequencies
- We still need to respect the jobs deadlines the best we can
- We can't go above the system maximum power consumption in any case

Given that we want the smallest possible power consumption despite making the best efforts to respect the deadlines - with the constraint of a system power cap - we established the following strategy to set the frequencies:

1. At each step we reinitialize all servers to their minimal frequencies
2. We fetch the servers which have at least one assigned job that is going to be late
3. We increase their frequencies with a step of 1 as long as we don't go over the system power cap and as long as there's still a job that is going to be late (and also as long as we haven't reached the maximum frequency allowed by the server)

This strategy led to the following implementation in the *ServersManager* class:

```
1   public void setFreqs(double date){
2       LinkedList<Server> lateServers = getLateServers(date);
3       boolean possibleIncFreq = !isOverMaxPow();
4
5       while(possibleIncFreq && !lateServers.isEmpty()){
6           for(Server s : lateServers){
7               int currFreq = 1;
8
9               while(s.willBeLate(date) && s.getCurrFreq() < s.getMaxFreq() && !
    isOverMaxPow()) {
10                  s.setCurrFreq(currFreq);
11                  currFreq++;
12              }
13
14              if(isOverMaxPow()){
15                  s.setCurrFreq(currFreq-1);
16                  possibleIncFreq = false;
17                  break;
18              }
19              else lateServers.remove(s);
20          }
21      }
22  }
```

Listing 21: Source code of the *setFreqs()* method

Last but not least, we have to be very careful about the number of units of work we decrement from the jobs on their respective servers. This is why on the *ServersManager* class we compute this amount across all servers based on each server current frequencies computed by the previously quoted method above. This gives us the following definition for the *decrementAll()* method:

```
1   public void decrementAll(double time){
2       for(Server s: servers){
3           if(s.isIdle()) continue;
4           // Units of work done with frequency 'freq' = time spent * 'freq':
5           s.getRunningJob().decrement(time * s.getCurrFreq());
6           if(s.getRunningJob().getUnitsOfWork() == 0) {
7               double start = ScheduleEntry.computeStart(scheduler.getSchedule()
    , s, s.getRunningJob());
8               double end = scheduler.getSchedule().currentDate;
9               ScheduleEntry newEntry = new ScheduleEntry(s.getRunningJob(), s,
    start, end, s.getCurrFreq());
10              scheduler.getSchedule().add(newEntry);
11              s.removeRunningJob();
12          }
13      }
14  }
```

Listing 22: Source code of the *decrementAll() method in the ServersManager class*

FIFO

3.12.1 Implementation

```
1     public void runScheduleStep() {
2         if(serversM.areAllServersIdle() && !arrivedJ.isEmpty()){
3             schedule.currentDate = arrivedJ.getFirst().getArrivalDate();
4             serversM.initServers();
5         }
6
7         //We start with frequencies at minimum:
8         serversM.resetFreqs();
9         //We increase frequencies if jobs are going to be late from the current
      date:
10        serversM.setFreqs(schedule.currentDate);
11
12        //We compute next event date:
13        double nextEventDate = getNextEventDate();
14        double duration = nextEventDate - schedule.currentDate;
15        schedule.currentDate += duration;
16
17        //We decrement and deal with finished jobs:
18        serversM.decrementAll(duration);
19
20        //We deal with new arrivals:
21        arrivedJ.addAll(jobsB.getArrivedJobs(nextEventDate));
22        assignArrivals();
23    }
```

Listing 23: Source code of an energy aware FIFO adapted for multiple servers

As you may notice, compared to the presentation done in part 1 of this report, a few additional lines have appeared in the schedule step dealing with servers frequencies. Actually, allowing servers to adapt their frequencies changes so much the output schedule that we could not factorize the code and we had to create a new *FIFOe* class for this energy aware FIFO.

3.12.2 Scheduling results

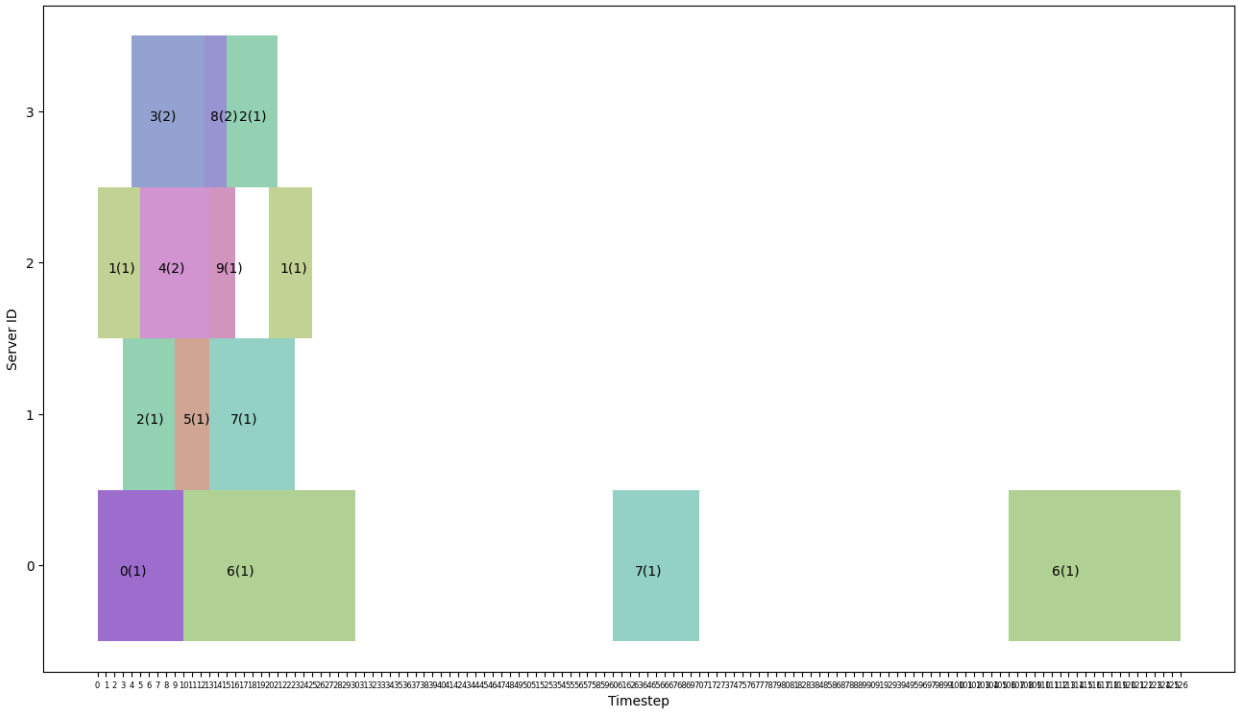

Figure 10: Output schedule produced by the FIFO scheduling algorithm on multiple servers and trying to use the smallest amount of power (with a power cap) while trying to not miss any deadline

Which corresponds to following file (<jobID, serverID, startDate, endDate, frequency>):

```
1 2 0.0 5.0 1.0
2 1 3.0 9.0 1.0
0 0 0.0 10.0 1.0
3 3 4.0 12.5 2.0
5 1 9.0 13.0 1.0
4 2 5.0 13.0 2.0
8 3 12.5 15.0 2.0
9 2 13.0 16.0 1.0
2 3 15.0 21.0 1.0
7 1 13.0 23.0 1.0
1 2 20.0 25.0 1.0
6 0 10.0 30.0 1.0
7 0 60.0 70.0 1.0
6 0 106.0 126.0 1.0
```

Listing 24: Detail of the scheduling result

3.12.3 Metrics

```
################## SCHEDULE METRICS ##################

>> Scheduling Metrics:
- Total Makespan: 126.0
- Nb Deadline Misses: 0
- Max Tardiness: 0.0
- Average Tardiness: 0.0
- Late Jobs:

>> Servers Metrics:
- Servers work load:
Server #0 : 60.0
Server #1 : 20.0
Server #2 : 21.0
Server #3 : 17.0

>> Energy Metrics:
- Total Consumption: 5833.3333333333485
- Max Consumption: 361.11111111111114
- Average Consumption: 1458.3333333333371
- Consumption per Server:
Server #0 : 1333.3333333333317
Server #1 : 1000.0
Server #2 : 999.9999999999998
Server #3 : 2500.0

#####################################################
```

Listing 25: Metrics for FIFO on multiple energy aware servers

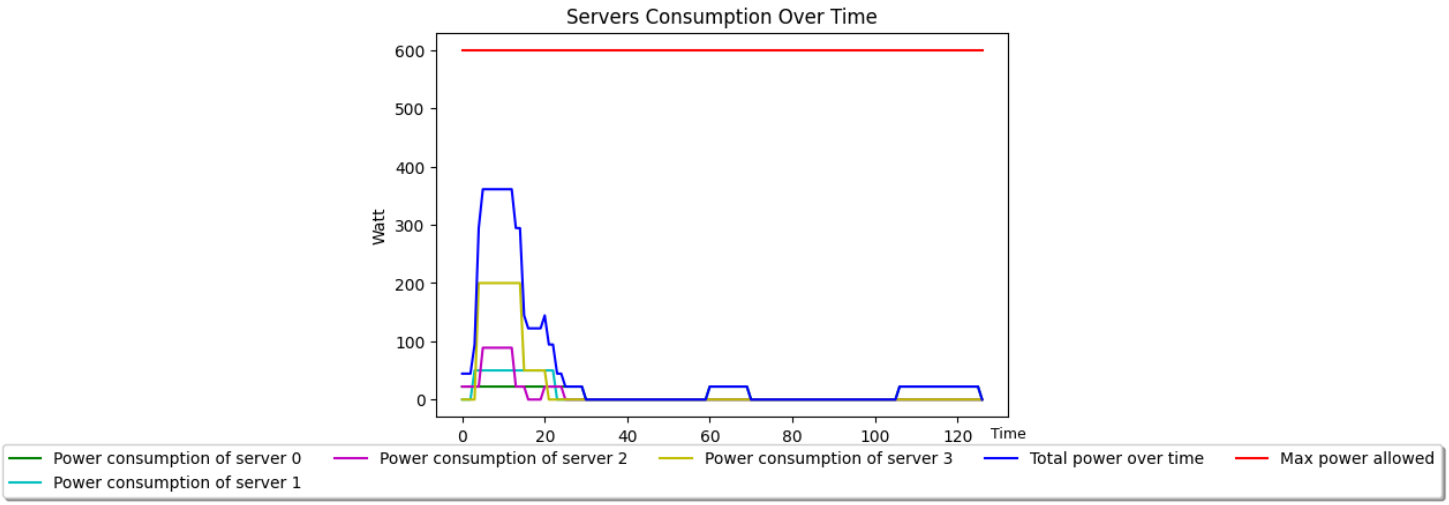

Figure 11: Energy consumption of FIFO servers through time

EDF

3.13.1 Implementation

```
1     protected void runScheduleStep() {
2         arrivedJ.sort(JOBS_COMPARISON_KEY);
3         if(serversM.areAllServersIdle() && !arrivedJ.isEmpty()){
4             schedule.currentDate = arrivedJ.getFirst().getArrivalDate();
5             serversM.initServers();
6         }
7
8         //We start with frequencies at minimum:
9         serversM.resetFreqs();
10        //We increase frequencies if jobs are going to be late from the current
      date:
11        serversM.setFreqs(schedule.currentDate);
12
13        //We compute next event date:
14        double nextEventDate = getNextEventDate();
15        double unitsOfWorkDone = nextEventDate - schedule.currentDate;
16        schedule.currentDate += unitsOfWorkDone;
17
18        //We decrement and deal with finished jobs:
19        serversM.decrementAll(unitsOfWorkDone);
20
21        //We deal with new arrivals:
22        arrivedJ.addAll(jobsB.getArrivedJobs(nextEventDate));
23        arrivedJ.sort(JOBS_COMPARISON_KEY);
24        assignArrivals(JOBS_COMPARISON_KEY, JOBS_COMPARISON_PREDICATE);
25    }
```

Listing 26: Source code of an energy aware EDF adapted for multiple servers

For the same reasons as with the energy aware FIFO presented in subsection 3.12.1 we had to create a new *EDFe* class that allows the servers to tune their frequencies. The comparison key/predicates follow the same logic as before.

3.13.2 Scheduling results

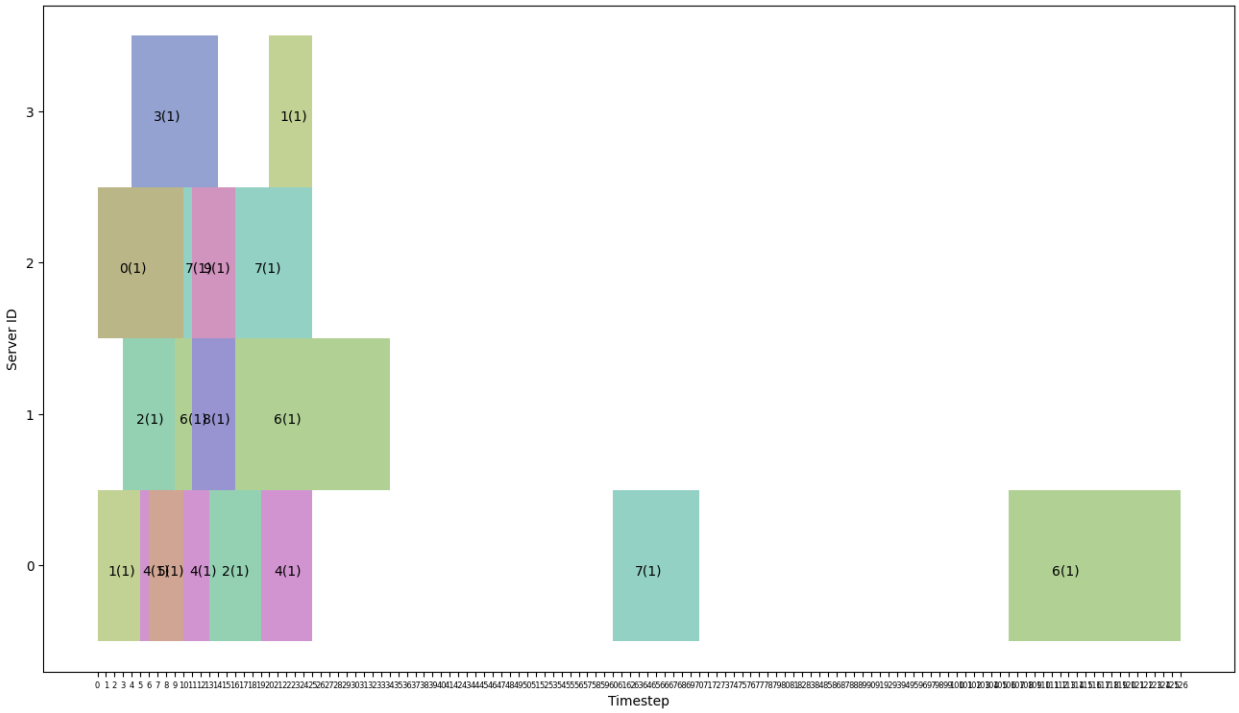

Figure 12: Output schedule produced by the EDF scheduling algorithm on multiple servers and trying to use the smallest amount of power (with a power cap) while trying to not miss any deadline

Which corresponds to following file (<jobID, serverID, startDate, endDate, frequency>):

```
1 0 0.0 5.0 1.0
4 0 5.0 6.0 1.0
2 1 3.0 9.0 1.0
5 0 6.0 10.0 1.0
0 2 0.0 10.0 1.0
6 1 9.0 11.0 1.0
7 2 10.0 11.0 1.0
4 0 10.0 13.0 1.0
3 3 4.0 14.0 1.0
8 1 11.0 16.0 1.0
9 2 11.0 16.0 1.0
2 0 13.0 19.0 1.0
4 0 19.0 25.0 1.0
7 2 16.0 25.0 1.0
1 3 20.0 25.0 1.0
6 1 16.0 34.0 1.0
7 0 60.0 70.0 1.0
6 0 106.0 126.0 1.0
```

Listing 27: Detail of the scheduling result

3.13.3 Metrics

```
################## SCHEDULE METRICS ##################

>> Scheduling Metrics:
- Total Makespan: 126.0
- Nb Deadline Misses: 0
- Max Tardiness: 0.0
- Average Tardiness: 0.0
- Late Jobs:

>> Servers Metrics:
- Servers work load:
Server #0 : 55.0
Server #1 : 31.0
Server #2 : 25.0
Server #3 : 15.0

>> Energy Metrics:
- Total Consumption: 4077.777777777775
- Max Consumption: 144.44444444444446
- Average Consumption: 1019.4444444444438
- Consumption per Server:
Server #0 : 1222.2222222222208
Server #1 : 1550.0
Server #2 : 555.5555555555555
Server #3 : 750.0

#####################################################
```

Listing 28: Metrics for EDF on multiple energy aware servers

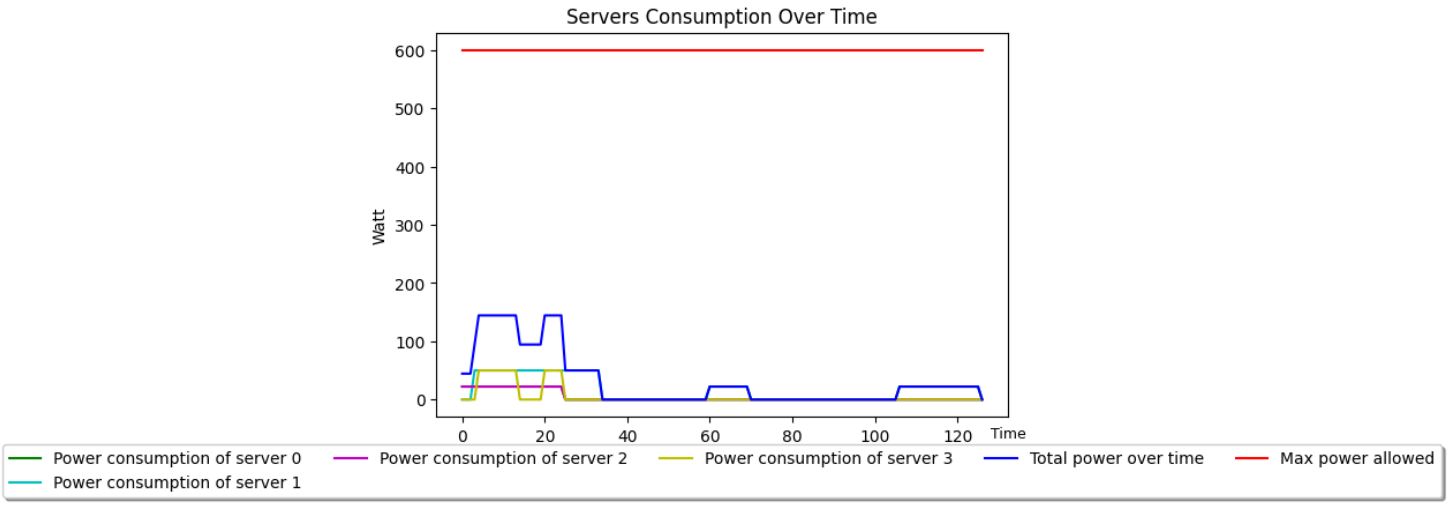

Figure 13: Energy consumption of EDF servers through time

Comparison Table

Algorithm	Pros	Cons	Possible Uses
FIFO	• Still easy to implement and maintain even across multiple servers	• Respecting the deadlines requires a lot of energy • Workload across servers isn't very well balanced • Still not starvation-free (but also the case for EDF...)	• If energy isn't a problem it's still the best algorithm to deal with tasks in their order of arrivals
EDF	• Lowest energy consumption as there was no deadline misses in the first place • Best servers workload balance	• Same as those mentioned in the multiserver part of this report	• Same as those mentioned in the multiserver part of this report