Dynamic Power Allocation in Server Farms: a Real Time Optimization Approach

ELCE 458: FINAL PROJECT

Sanzhar Kaidarov, Aslan Sabyrov

A report contains a simple explanation of work done that is easy to understand and has many simplifications made due to the difficulty of the problem space. In short, the aim of this final project is to find a best suboptimal solution to the power allocation in server farm such that the server farm still meets performance constraints.

Problem Statement

Server farm has some *c* amount of servers and jobs that need to be processed are arriving at the server farm. In the article we define job arrival as a random process, intuitively it is true – we don't know if at some point we are going to receive a lot of jobs or no jobs at all, and mostly jobs are not dependent on each other. Trusting the authors of the article, the random process is a Poisson process, but although the arrivals of jobs are independent, we assume that the arrival rate follows some pattern which we modeled as a Markov Modulated Poisson Process.

In short, MMPP (Markov Modulated Poisson Process) is a process where you have some states, and you can transition between states with some probability. Sum of transitions from any given state would be equal to 1, and all of the transitions are compiled in a transition matrix P. An important feature of MMPP is that we can try to guess the next state by considering a transition matrix. This would be useful later when we attempt to predict the next state and consequently a work load.

In our case, jobs can be modeled adhering to three states - idle, normal and busy. It makes sense because we can imagine that some of the time there are more incoming jobs to process (for example the start of a share market in the morning), and some of the time there are less incoming jobs to process (when the share market is closed). We can transition from idle to normal, stay in the idle, transition from normal to idle or busy or stay normal and transition from busy to normal or stay busy as can be seen in the Fig 1.

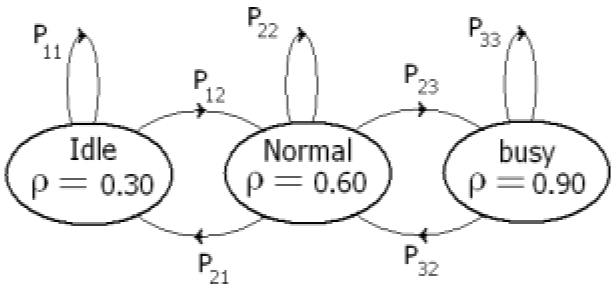


Figure 1.Markov chain rate in incoming job MMPP model

Now that we modeled our arriving jobs we can continue to the actual problem - we want to allocate power to each server in an efficient manner.

There are two power allocation schemes: static and dynamic. For static allocation we do not care about the time series, meaning we give each server some amount of power independent of if there is a high load or no load. The focus of our work is a dynamic power allocation.

Dynamic power allocation tries to adjust the power given to the servers according to the prediction of future demand on servers. For example, if we get some number X[i] of jobs right now, and we know the distribution of X incoming jobs, using the knowledge of the nature of the distribution we can have an intelligent guess of the X[i+1] number of jobs at the next time slot. This prediction will make our decision making process adjust the number of servers that would be **on** during time (i+1).

Our decision making process is defined as a variable *u* which is a signed number that would show us the command to perform (turn *u* servers on, or if *u* is negative, turn *u* servers off). However when sending a command to turn a server on, it takes some time *Ts* before it is active. We assume discrete time sampling for our convenience and the number of jobs that a server can process at a time is therefore a unit time (1).

Finally the decision making variable *u* is a variable that would be an input to some *E(u)* cost function that represents the total power consumption of our dynamic power allocation scheme considering the decision *u* that we make upon a prediction based on our knowledge of the distribution. Therefore, our optimization process is a minimization problem of our cost function *E*.

Now the constraints that we want to meet is that because our switching command *u* is not applied instantaneously, there is delay in time of when we can process some jobs. We want to keep the delay due to *Ts* start up time lower than some T_D delay that we estimate at a steady state.

Finally, the optimization process can be rewritten as:

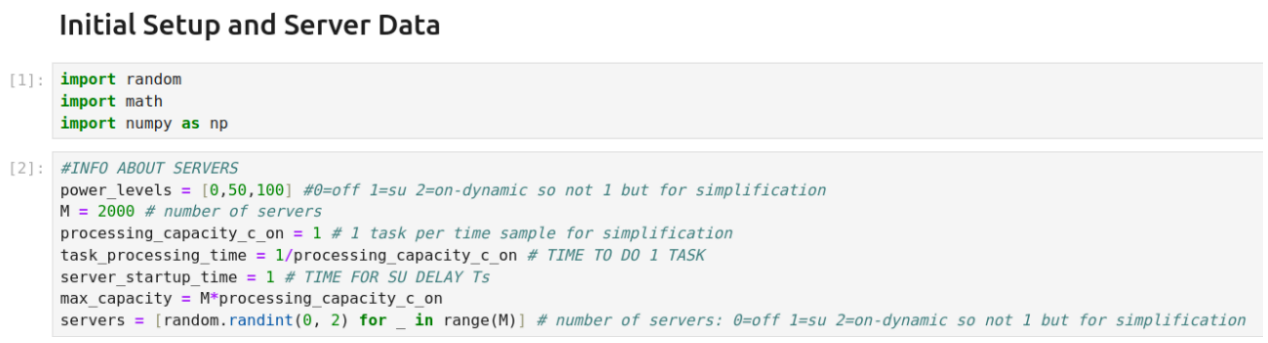
$$\begin{gathered} \min_{u} \quad E(u) \\ D(u) \leq T_D. \end{gathered}$$
,

where D(u) is a delay due to *Ts* and T_D is a delay at a steady state.

Implementation

Data cleaning and establishing constants

First important steps and to write out all the assumed parameters either by us or by the paper researchers. Most of these are self-explanatory and are simple to understand like power levels (which corresponds to power drawn if a server is on, is starting up or is off), processing capacity per unit time, maximum capacity of our farm, number of servers and the state of the servers at the start:



Initial Setup and Server Data

```
[1]: import random
     import math
     import numpy as np
```

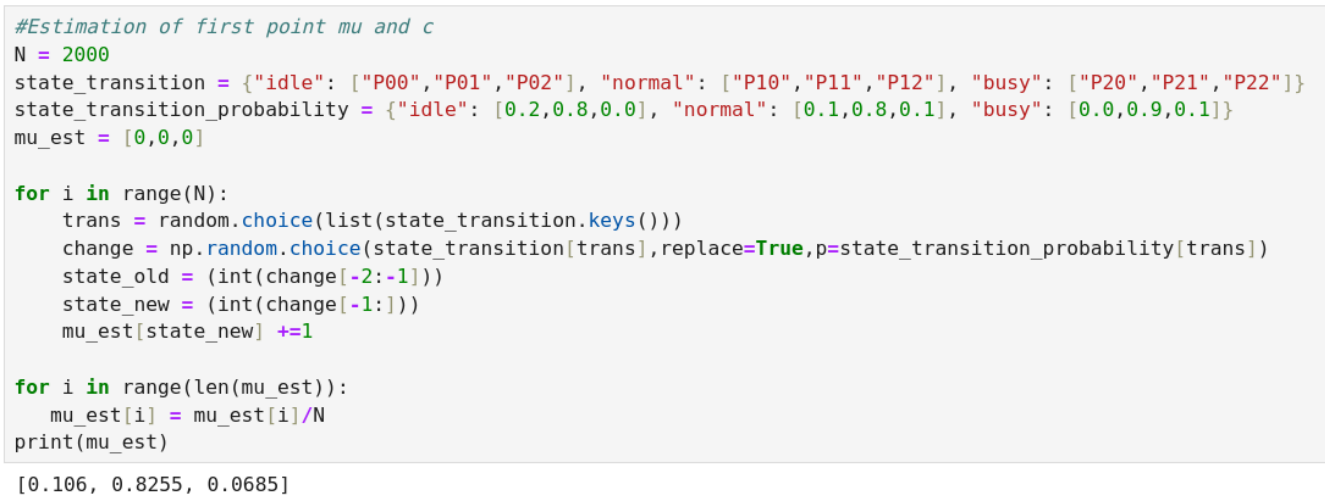
```
[2]: #INFO ABOUT SERVERS
     power_levels = [0,50,100] #0=off 1=su 2=on-dynamic so not 1 but for simplification
     M = 2000 # number of servers
     processing_capacity_c_on = 1 # 1 task per time sample for simplification
     task_processing_time = 1/processing_capacity_c_on # TIME TO DO 1 TASK
     server_startup_time = 1 # TIME FOR SU DELAY Ts
     max_capacity = M*processing_capacity_c_on
     servers = [random.randint(0, 2) for _ in range(M)] # number of servers: 0=off 1=su 2=on-dynamic so not 1 but for simplification
```

Figure 1. Data establishment.

Estimation of starting point mu and server states

Next we estimate a starting point. It is done quite simply and robustly, providing us with some point close to the mu_opt and server_opt for the mean of the value data after 2000 iterations. It also clearly shows one step of how MMMP works.

So, to model MMPP in Python we create a dictionary with states, we create transition probabilities and model a transition matrix. Using a np.random.choice we make a random choice with respect to the transition matrix (if the probability of transition to **normal state** is highest, we would expect a biased random choice towards the **normal state**). We look at the transitions and see what state transitions to what state in some N iterations. It is important to note that we have biased the state_transition_probability matrix towards normal state, as you can see paths towards “normal” state weight more, while “idle” and “busy” are less likely to happen in our model and understanding of the server farm.



```
#Estimation of first point mu and c
N = 2000
state_transition = {"idle": ["P00","P01","P02"], "normal": ["P10","P11","P12"], "busy": ["P20","P21","P22"]}
state_transition_probability = {"idle": [0.2,0.8,0.0], "normal": [0.1,0.8,0.1], "busy": [0.0,0.9,0.1]}
mu_est = [0,0,0]

for i in range(N):
    trans = random.choice(list(state_transition.keys()))
    change = np.random.choice(state_transition[trans],replace=True,p=state_transition_probability[trans])
    state_old = (int(change[-2:-1]))
    state_new = (int(change[-1:]))
    mu_est[state_new] +=1

for i in range(len(mu_est)):
   mu_est[i] = mu_est[i]/N
print(mu_est)
```

```
[0.106, 0.8255, 0.0685]
```

Figure 2. MMPP model considering random starting state

RTO Algorithm

Now we define our vectors and we try to find our steady state optimal set points. An optimal set point is a complex solution to the Real Time Optimization approach that involves finding zeros of a Probability Generating Function. Logically we assumed that in a long run, where prediction of jobs at a server at time k and the number of arrived of the jobs at k would converge to the same value which is a rate of arrival . Then we calculate our switching controller decision variable *u* as this:

$$u = \left\lfloor \left(\alpha(\hat{c} - \bar{c}) + \beta(\hat{l} - \bar{l}) \right) \right\rfloor$$

Now our prediction for number of jobs at a server l is:

$$l(k+1) = l(k) + X(k) - min\left\{ l(k), c^u(k) \right\},$$

where c u (k) is the service rate of the server farm at the time k when the switching command control u is applied.

As for c, we take an assumption that for a fixed input rate λ, the optimal (most economic) valid service rate is the smallest value of ¯c for which the stability condition holds (¯c > λ) which is basically a ceiling function for λ.

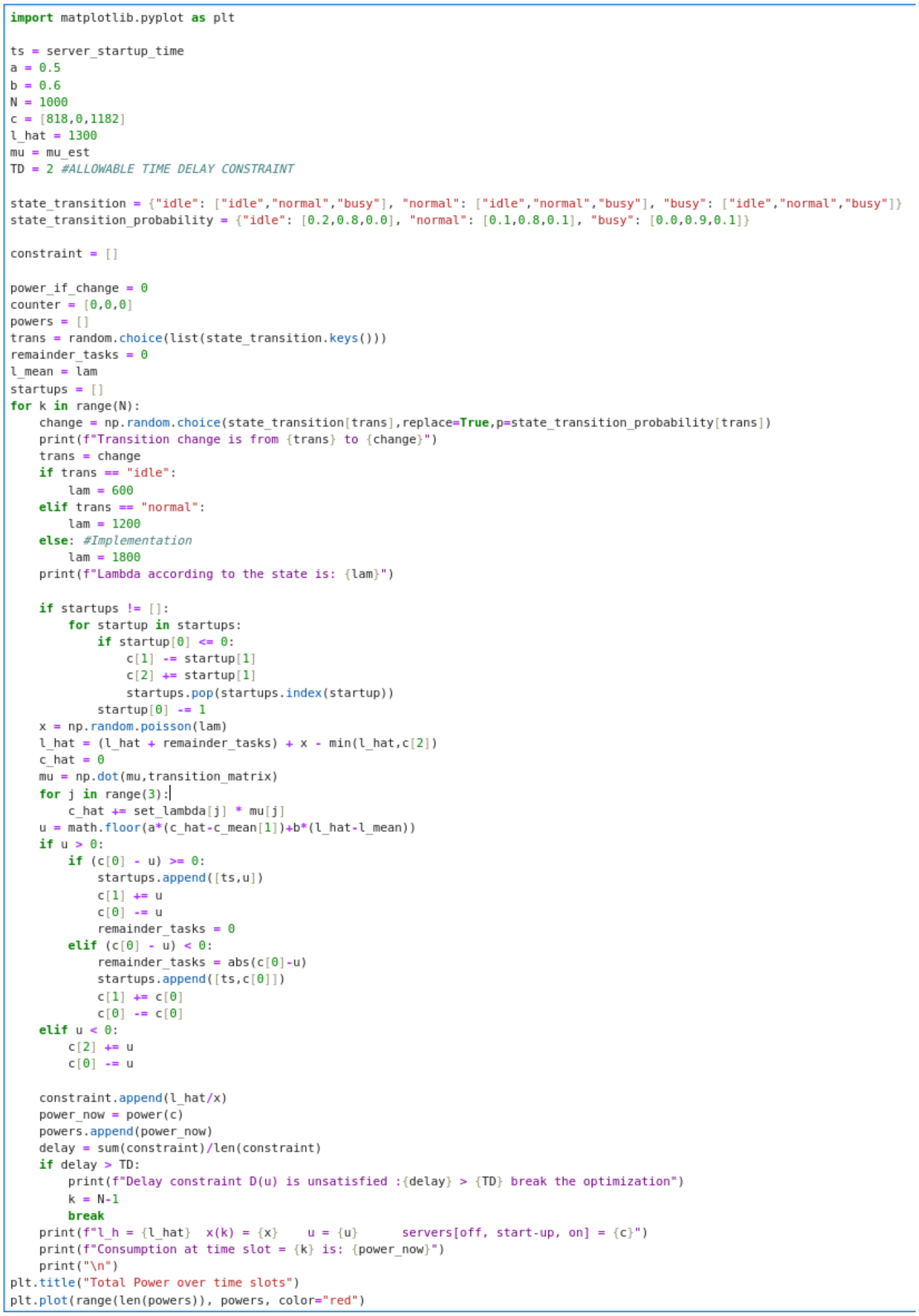
The prediction for c we assumed to be a prediction of ceiling ¯λ which is:

$$\begin{aligned} \hat{\lambda}(k+T_s|k) &= \mathbb{E}[X(k+T_s)] \qquad (18) \\ &= \sum_{j=1}^{n} \mathbb{E}[X(k+T_s)|Z(k+T_s)=j]\, \mathbb{P}[Z(k+T_s)=j] \\ &= \sum_{j=1}^{n} \lambda_j\, \mu_j(k+T_s|k), \end{aligned}$$

where the probability distribution $\boldsymbol{\mu}(k+T_s|k)$ can be computed as

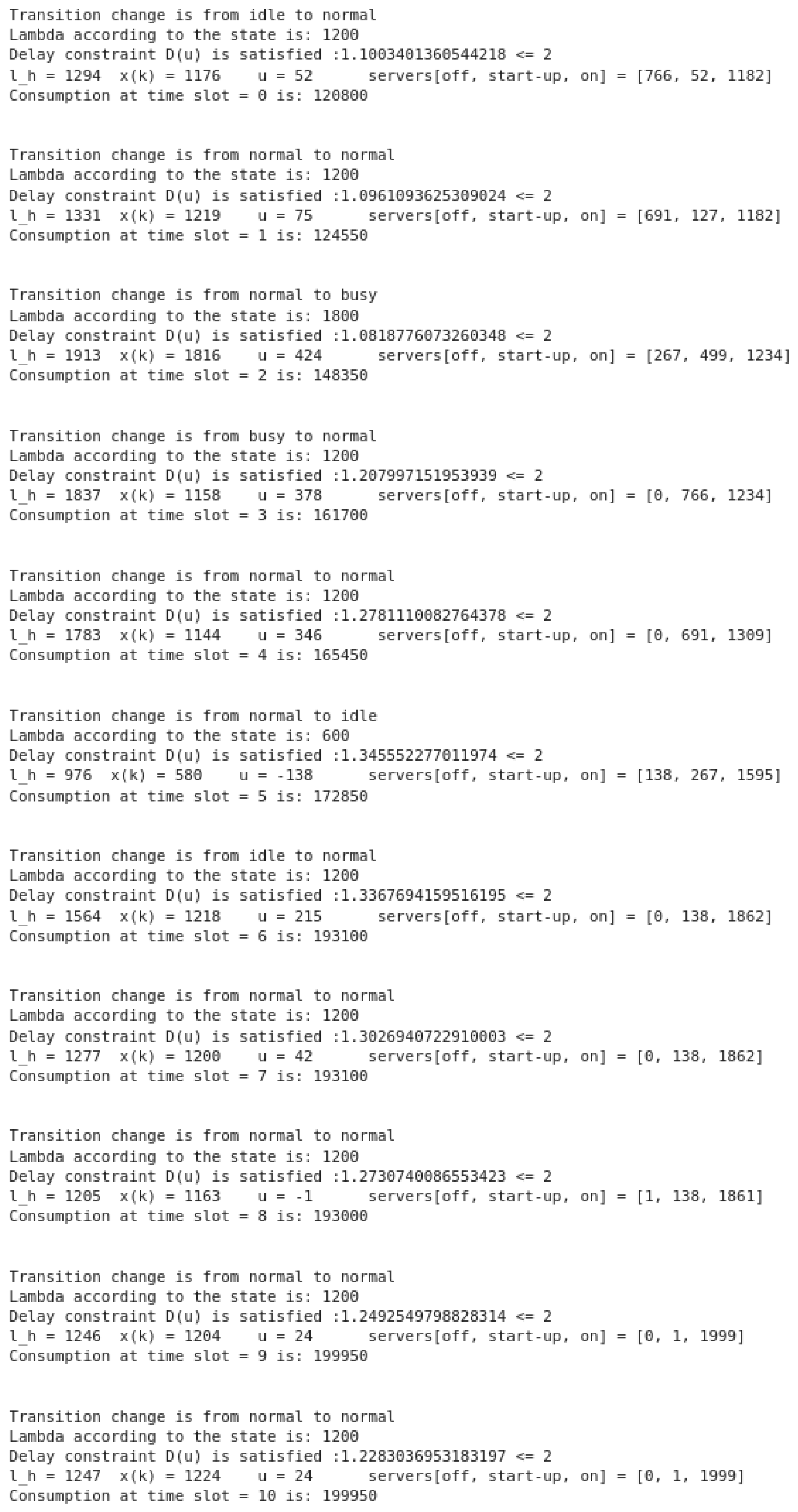
$$\boldsymbol{\mu}(k+T_s|k) = \boldsymbol{\mu}(k|k)P^{T_s}. \qquad (19)$$

We implemented this part of this work in Fig.3



```python
import matplotlib.pyplot as plt

ts = server_startup_time
a = 0.5
b = 0.6
N = 1000
c = [818,0,1182]
l_hat = 1300
mu = mu_est
TD = 2 #ALLOWABLE TIME DELAY CONSTRAINT

state_transition = {"idle": ["idle","normal","busy"], "normal": ["idle","normal","busy"], "busy": ["idle","normal","busy"]}
state_transition_probability = {"idle": [0.2,0.8,0.0], "normal": [0.1,0.8,0.1], "busy": [0.0,0.9,0.1]}

constraint = []

power_if_change = 0
counter = [0,0,0]
powers = []
trans = random.choice(list(state_transition.keys()))
remainder_tasks = 0
l_mean = lam
startups = []
for k in range(N):
    change = np.random.choice(state_transition[trans],replace=True,p=state_transition_probability[trans])
    print(f"Transition change is from {trans} to {change}")
    trans = change
    if trans == "idle":
        lam = 600
    elif trans == "normal":
        lam = 1200
    else: #Implementation
        lam = 1800
    print(f"Lambda according to the state is: {lam}")

    if startups != []:
        for startup in startups:
            if startup[0] <= 0:
                c[1] -= startup[1]
                c[2] += startup[1]
                startups.pop(startups.index(startup))
            startup[0] -= 1
    x = np.random.poisson(lam)
    l_hat = (l_hat + remainder_tasks) + x - min(l_hat,c[2])
    c_hat = 0
    mu = np.dot(mu,transition_matrix)
    for j in range(3):
        c_hat += set_lambda[j] * mu[j]
    u = math.floor(a*(c_hat-c_mean[1])+b*(l_hat-l_mean))
    if u > 0:
        if (c[0] - u) >= 0:
            startups.append([ts,u])
            c[1] += u
            c[0] -= u
            remainder_tasks = 0
        elif (c[0] - u) < 0:
            remainder_tasks = abs(c[0]-u)
            startups.append([ts,c[0]])
            c[1] += c[0]
            c[0] -= c[0]
    elif u < 0:
        c[2] += u
        c[0] -= u

    constraint.append(l_hat/x)
    power_now = power(c)
    powers.append(power_now)
    delay = sum(constraint)/len(constraint)
    if delay > TD:
        print(f"Delay constraint D(u) is unsatisfied :{delay} > {TD} break the optimization")
        k = N-1
        break
    print(f"l_h = {l_hat}  x(k) = {x}    u = {u}      servers[off, start-up, on] = {c}")
    print(f"Consumption at time slot = {k} is: {power_now}")
    print("\n")
plt.title("Total Power over time slots")
plt.plot(range(len(powers)), powers, color="red")
```

Fig.3 RTO Algorithm in Python



```
Transition change is from idle to normal
Lambda according to the state is: 1200
Delay constraint D(u) is satisfied :1.1003401360544218 <= 2
l_h = 1294  x(k) = 1176    u = 52      servers[off, start-up, on] = [766, 52, 1182]
Consumption at time slot = 0 is: 120800


Transition change is from normal to normal
Lambda according to the state is: 1200
Delay constraint D(u) is satisfied :1.0961093625309024 <= 2
l_h = 1331  x(k) = 1219    u = 75      servers[off, start-up, on] = [691, 127, 1182]
Consumption at time slot = 1 is: 124550


Transition change is from normal to busy
Lambda according to the state is: 1800
Delay constraint D(u) is satisfied :1.0818776073260348 <= 2
l_h = 1913  x(k) = 1816    u = 424      servers[off, start-up, on] = [267, 499, 1234]
Consumption at time slot = 2 is: 148350


Transition change is from busy to normal
Lambda according to the state is: 1200
Delay constraint D(u) is satisfied :1.207997151953939 <= 2
l_h = 1837  x(k) = 1158    u = 378      servers[off, start-up, on] = [0, 766, 1234]
Consumption at time slot = 3 is: 161700


Transition change is from normal to normal
Lambda according to the state is: 1200
Delay constraint D(u) is satisfied :1.2781110082764378 <= 2
l_h = 1783  x(k) = 1144    u = 346      servers[off, start-up, on] = [0, 691, 1309]
Consumption at time slot = 4 is: 165450


Transition change is from normal to idle
Lambda according to the state is: 600
Delay constraint D(u) is satisfied :1.345552277011974 <= 2
l_h = 976  x(k) = 580    u = -138      servers[off, start-up, on] = [138, 267, 1595]
Consumption at time slot = 5 is: 172850


Transition change is from idle to normal
Lambda according to the state is: 1200
Delay constraint D(u) is satisfied :1.3367694159516195 <= 2
l_h = 1564  x(k) = 1218    u = 215      servers[off, start-up, on] = [0, 138, 1862]
Consumption at time slot = 6 is: 193100


Transition change is from normal to normal
Lambda according to the state is: 1200
Delay constraint D(u) is satisfied :1.3026940722910003 <= 2
l_h = 1277  x(k) = 1200    u = 42      servers[off, start-up, on] = [0, 138, 1862]
Consumption at time slot = 7 is: 193100


Transition change is from normal to normal
Lambda according to the state is: 1200
Delay constraint D(u) is satisfied :1.2730740086553423 <= 2
l_h = 1205  x(k) = 1163    u = -1      servers[off, start-up, on] = [1, 138, 1861]
Consumption at time slot = 8 is: 193000


Transition change is from normal to normal
Lambda according to the state is: 1200
Delay constraint D(u) is satisfied :1.2492549798828314 <= 2
l_h = 1246  x(k) = 1204    u = 24      servers[off, start-up, on] = [0, 1, 1999]
Consumption at time slot = 9 is: 199950


Transition change is from normal to normal
Lambda according to the state is: 1200
Delay constraint D(u) is satisfied :1.2283036953183197 <= 2
l_h = 1247  x(k) = 1224    u = 24      servers[off, start-up, on] = [0, 1, 1999]
Consumption at time slot = 10 is: 199950
```

Fig.4 Results of some iterations after RTO Algorithm…

As we can see analytically, with the change of u we switch servers on, after Ts time it will be on, but before Ts has passed it is buffered as a SU state. In our case c is the combination of all servers with c[0] being the number of servers OFF, c[1] number of servers on SU, and c[2] number of servers ON.

According to the State Transition Matrix, every k time slot there is some transition happening, be it state comes back to itself or goes to different states.

The Time delay is also implemented. As said in the paper, it is a constraint that does not allow for taks to accumulate more than a certain time TD threshold(2s in our case). Most of the time our simulation is terminated when time delay is not satisfied, because this optimization approach is very prone to sharp data changes and reacts well only to small data deviations(like Poisson distribution)

There are a lot of numerical difficulties in modeling and implementing the solution, and also justifying the argument that the original authors did - assuming a slowly varying rate of arrival of jobs, dynamic approach can prove beneficial, however if the processes show unexpected value, the dynamic approach would degrade very fast. Which we believe is the biggest limitation of this approach. Here the lucky case in Fig.5, you can see that although the power change was significant by the peaks, the algorithm still handled it well. It is also interesting to see the "Real Time" Optimization, as the data is random, our goal is to minimize it on every iteration, and on the graph you can see that after every peak, the power consumption goes down and is getting optimized.

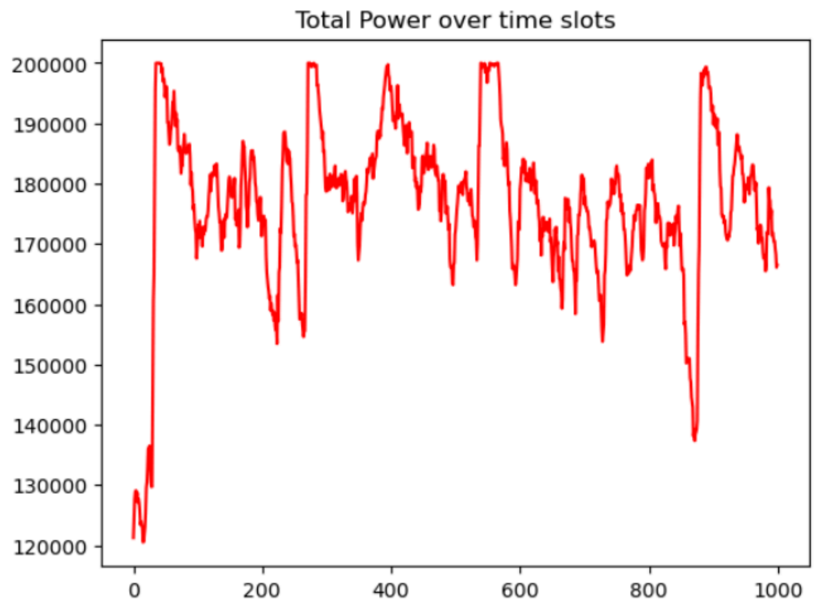


Fig.5 "Lucky" Case of RTO Approach for Ts(startup time)=1k

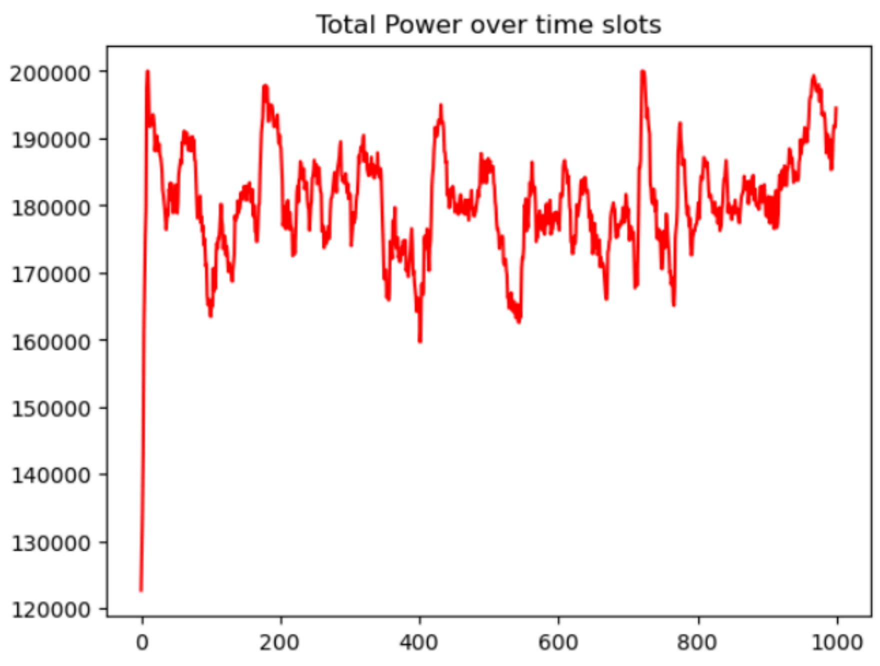


Fig.6 “Lucky” Case of RTO Approach for Ts(startup time)=2k

However, looking at the case when the algorithm fails, is when the constraint of time delay is not satisfied, which means that even though the server farm is running on its max capacity(all servers are on), it can’t handle all requests and there is big data overflow. REMARK, when we tried to make an algorithm fail, usually it did it constantly by the end of 1000 iterations. However, while making the code better and cleaner, we somehow optimized it too well, so the time delay actually never becomes bigger than 1.08k for startup time = 1s.

There are also a lot of limitations to what the paper explains on prediction, and we heavily rely on our assumptions about PGF that make some sense and are confirmed by the code. We implemented the algorithm successfully and believe it works and optimizes in a very good manner.