

# Cloud Computing Assignment 1

Submitted By – Jui Morey

ID – 213is002

**Paper Title** - METO: Matching-Theory-Based Efficient Task Offloading in IoT-Fog Interconnection Networks.

**Paper Objective / Basic Understanding –**

This paper is basically about fog computing for IOT devices for a network with stringent service delay requirements.

For applications which require huge processing power and lots of data is involved such applications cannot be executed on the IOT device. In traditional cloud computing scenario such applications have to be computed on the cloud. But for applications involving AI , augmented reality, etc the time delay would be huge. So fog computing can be used to reduce this delay. In fog computing intermediate devices called fog devices are introduced between cloud and IOT devices. The computing would be done at this fog devices which will in turn reduce the latency. But in a practical scenario when each IOT devices generates tasks then we aim to reduce the total latency and energy consumed by mapping the tasks to appropriate fog nodes.

This paper gives an algorithm called METO which aims to reduce the latency and energy consumption of IOT and fog devices.

## Algorithm Implementation-

Simulator Used – Ifogsim.

This algorithm has four stages -

1. Decision Matrix Formulation
2. CRITIC
3. TOPSIS
4. Matching

They are implemented as follows-

### 1. Decision Matrix Formulation-

```
2. private static void createDecisionMatrix(){
3.     int noOfFN=fogDevices.size();
4.     int noOfTask=tasks.size();
5.
6.     // creating decision matrix for Tasks of IOT devices
7.     // matrix dTask size = noOfFN x 2
8.
9.     for(int k=0;k<noOfTask;k++){
10.        Vector<Vector<float>> dTask= new
Vector<Vector<float>>(noOfFN);
11.        for(int i=0;i<noOfFN;i++){
12.            Vector<float> row=new Vector<float>(2);
13.            row[0]=fogDevices[i].latency;
14.            row[1]=fogDevices[i].energyConsumption;
15.            dTask[i]=row;
16.        }
17.        decisionMatrix.add(dTask);
18.    }
19.
20.
21.    // creating decision matrix for FN
22.    // matrix dFN size = noOfTask x 2
23.    for(int k=0;k<noOfFN;k++){
24.        Vector<Vector<float>> dFN= new
Vector<Vector<float>>(noOfTask);
25.        for(int i=0;i<noOfTask;i++){
26.            Vector<float> row=new vector<float>(2);
27.            row[0]=tasks[i].latency;
```

```

28.             row[1]=tasks[i].energyConsumption;
29.             dFN[i]=row;
30.         }
31.         decisionMatrix.add(dFN);
32.     }
33.
34. }

```

The decision matrix has to be formulated for each task and each fog node. For a task the decision matrix will be of size  $n \times 2$  where  $n$  is the no. of fog nodes and two columns are required to enter the values of two parameters latency and energy of each IOT device.

For a fog node the decision matrix is of size  $m \times 2$  where  $m$  is no. of tasks and two columns are required to enter values of two criterion energy consumed by fog devices to execute that task and deadline of the task.

## 2. CRITIC-

```

private static void CRITIC(){

    int m=decisionMatrix.size();//m= np. of task nodes + no. of fog nodes

    for(int l=0;l<m;l++){
        Vector<Vector<float>> B=decisionMatrix[l]; //taking each agent 'a'
belongs to A=T U F
        int n=B.size();
        int c=B[0].size();

        // Normalize the decision matrix of an agent 'a' as per Eq. (16).
        Vector<float> best=new Vector<float>(c);
        Vector<float> worst=new Vector<float>(c);
        for(int j=0;j<c;j++){
            best[j]=B[0][j];
            worst[j]=B[0][j];
            for(int i=0;i<n;i++){
                best[j]=max(best[j],B[i][j]);
                worst[j]=min(worst[j],B[i][j]);
            }
        }
    }
}

```

```

    }
}
for(int j=0;j<c;j++){
    for(int i=0;i<n;i++){
        B[i][j]=(B[i][j]-worst[j])/(best[j]-best[j]); //
Executing Eq 16
    }
}

//Evaluate the standard deviation  $\sigma_k$  for each criterion in the
normalized decision matrix
Vector<float> SD=new Vector<float>(c);
for(int j=0;j<c;j++){
    Vector<float> v=new Vector<float>(n);
    for(int i=0;i<n;i++) v[i]=B[i][j];
    SD[j]=calculateSD(v);
}

// constructing a criteria correlation matrix which is a symmetric
matrix of size c*c

Vector<Vector<float>> S=new Vector<Vector<float>>(c);
for(int i=0;i<c;i++){
    S[i]=new Vector<float>(c);
    for(int j=0;j<c;j++){
        Vector<float> v1=new Vector<float>(c);
        Vector<float> v2=new Vector<float>(c);
        for(int x=0;x<n;x++) v1[x]=(B[x][i]);
        for(int x=0;x<n;x++) v2[x]=(B[x][j]);
        S[i][j]=find_coefficient(v1,v2);
    }
}

// Determine each criterion weight  $w_k$  calculated as per Eq. (18)
and form the criteria weight vector  $W_a$  for agent 'a'
Vector<float> w=new Vector<float>(c);
float total_sum=0;
for(int i=0;i<c;i++){
    float sum=0;
    for(int j=0;j<n;j++)
        sum+=(1-S[i][j]);
    w[i]=SD[i]*sum; // Executing Eq (17)
}
for(int i=0;i<c;i++)
    total_sum+=w[i];

```

```

        for(int i=0;i<c;i++)
            w[i]=(w[i]/total_sum);    // Executing Eq (18)

        //Add Wa as next row in W.
        weightMatrix.add(w);
    }
}

```

The critic approach implemented above takes as input the decision matrix and computes a weight vector for each task and fog node and adds this weight vector to a matrix and we finally get a weight matrix. The weight matrix is computed as follows-

First normalize each value in a decision matrix for each criterion (ex- latency).

Then we calculate standard deviation for that criterion.

Before calculating the quantity of information we need a criteria correlation matrix which is generated as follows-

```

Vector<Vector<float>> S=new Vector<Vector<float>>(c);
    for(int i=0;i<c;i++){
        S[i]=new Vector<float>(c);
        for(int j=0;j<c;j++){
            Vector<float> v1=new Vector<float>(c);
            Vector<float> v2=new Vector<float>(c);
            for(int x=0;x<n;x++) v1[x]=(B[x][i]);
            for(int x=0;x<n;x++) v2[x]=(B[x][j]);
            S[i][j]=find_coefficient(v1,v2);
        }
    }

```

Next quantity of information is calculated by multiplying standard deviation of that criterion with 1-correlation coefficient.

Next we calculate the weight vector which consists of the weight values of each criterion. Weight value of each criterion is calculated by dividing the quantity of information for that criterion with sum of quantity of information for each criterion.

### 3. TOPSIS

```
private static void TOPSIS(){
    int m=decisionMatrix.size();

    for(int l=0;l<m;l++){
        Vector<Vector<float>> B=decisionMatrix[l]; //taking each agent 'a'
belongs to A=T U F
        int n=B.size();
        int c=B[0].size();

        //Normalize the decision matrix using Eq. (19).
        for(int j=0;j<c;j++){
            float square_sum=0;
            for(int i=0;i<n;i++) square_sum+=B[i][j]*B[i][j];
            square_sum=sqrt(square_sum);
            for(int i=0;i<n;i++) B[i][j]=(B[i][j]/square_sum); // Eq.
(19).
        }

        // Calculate the weighted normalized decision matrix based on Eq.
(20)
        for(int j=0;j<c;j++){
            for(int i=0;i<n;i++) B[i][j]=B[i][j]*weightMatrix[l][j]; //
Eq. (20)
        }

        Vector<float> positive=new Vector<float>(c);
        Vector<float> negative=new Vector<float>(c);
        // Evaluate the positive and negative ideal solutions for each
criterion k as
        // minimum and maximum value of kth column weighted normalized
matrix.
        for(int j=0;j<c;j++){
            negative[j]=positive[j]=B[0][j];
            for(int i=0;i<n;i++){
```

```

        negative[j]=max(negative[j],B[i][j]);
        positive[j]=min(positive[j],B[i][j]);
    }
}

Vector<float> d_pos=new Vector<float>(n);
Vector<float> d_neg=new Vector<float>(n);
Vector<float> p=new Vector<float>(n);
// Determine the distance of each alternative from the positive
and negative ideal
// solutions based on Eq.(21) and (22).
for(int i=0;i<n;i++){
    for(int j=0;j<c;j++){
        d_pos[i]+=(B[i][j]-positive[j])*(B[i][j]-positive[j]);
        d_neg[i]+=(B[i][j]-negative[j])*(B[i][j]-negative[j]);
    }
    d_pos[i]=sqrt(d_pos[i]); // Eq.(21)
    d_neg[i]=sqrt(d_neg[i]); // Eq.(22)
}

// Compute the performance score for each alternative following
Eq. (23),
for(int i=0;i<n;i++){
    p[i]=d_neg[i]/(d_neg[i]+d_pos[i]); // Eq.(23)
}

// ranked in decreasing order of their performance scores
sort(p.begin(),p.end());
reverse(p.begin(),p.end());

performance.add(p);
}
}

```

We take as input the decision matrix of all tasks and all fog nodes and compute the performance matrix as follows-

1. Normalize each entry in the decision matrix by dividing that entry by taking square root of sum of



squares of all entries corresponding to that criterion.

2. Calculate the weighted normalized decision matrix by multiplying the weight vector with normalized values.
3. Evaluate the positive and negative ideal solutions for each criterion  $k$  as minimum and maximum value of  $k$ th column weighted normalized matrix.
4. For each row in the weighted normalized decision matrix calculate two values  $d_1$  and  $d_2$  by using the commonly know distance formula for finding distance between two points.
5. Compute the performance score for each alternative by dividing  $d_2$  with sum of  $d_1$  and  $d_2$  and rank them in decreasing order.
6. Finally the performance matrix of each task and fog node is ready.

## 4 Proposed Task Offloading Algorithm (METO)

```
public static void matching(){  
  
    // Q -- Quota of each FN  
    Vector<Integer> Q=new Vector<Integer>(no_of_FN);  
  
    // Assign -- Assigned tasks in FN  
    Vector<Vector<Integer>> Assign=new Vector<Vector<Integer>>(no_of_FN);
```

```

        for(vector<float> tj:performance){
            int n=tj.size();
            for(int x=0;x<n;x++){
                // fi*= highest ranked FN in P(tj) to which tj has not
proposed yet
                // Send proposal to fi*.
                int fi=tj[x];
                if(Q[fi]>0){ // if Qi* > 0 then
                    Assign[fi].push_back(tj);
                    Q[fi]=Q[fi]-1;
                }
                else{
                    // Reject the assignment request;
                }
            }
        }
    }
}

```

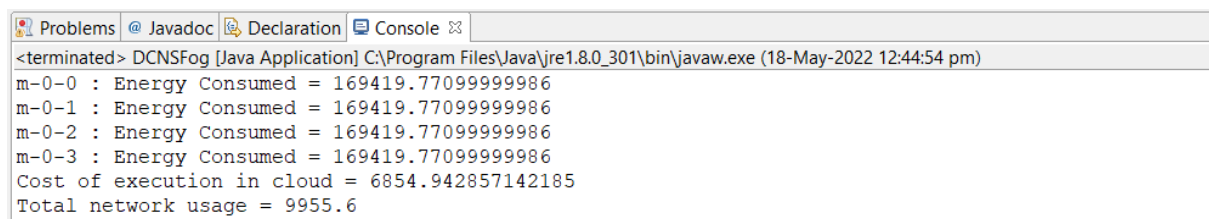
This algorithm gives the final mapping between tasks and fog devices that is which task should be assigned to which fog node such that latency and energy are minimized.

To do this mapping we take as input the performance matrix of each task and fog node.

For a task which is not assigned to any fog node, choose the fog node with maximum performance score and not yet selected. If a virtual resource unit is available then assign the task to that fog node and if the VRU is not available then find a task with less priority than the current task and if such a task is found then remove that task from that fog node and assign the current task. If such a task is not found then

choose the next best fog node for the task from the performance matrix and repeat the process.

## Predicted Output-



```
<terminated> DCNSFog [Java Application] C:\Program Files\Java\jre1.8.0_301\bin\javaw.exe (18-May-2022 12:44:54 pm)
m-0-0 : Energy Consumed = 169419.770999999986
m-0-1 : Energy Consumed = 169419.770999999986
m-0-2 : Energy Consumed = 169419.770999999986
m-0-3 : Energy Consumed = 169419.770999999986
Cost of execution in cloud = 6854.942857142185
Total network usage = 9955.6
```