Algorithmic Logic and Mathematical correlation:

Def Algo(workload, latency, RAN_loc):

Latency:
T = d/v
3* 10^8
0.01ms - 0.03ms - (double)
D = 3km - 9km - (int)

Workload:
140 - 150 task - (int)
Each RAN - Task limit - 2
MEC Task limit = Workload_cap
Max number in the cluster can be = Wokload_cap/RAN_cap

……..
……..
……..
……..

Naive Based K means Implementation:

```
import random
import math

#Euclidian Distance between two d-dimensional points
def eucldist(p0,p1):
    dist = 0.0
    for i in range(0,len(p0)):
        dist += (p0[i] - p1[i])**2
    return math.sqrt(dist)



#K-Means Algorithm
def kmeans(k,datapoints):

    # d - Dimensionality of Datapoints
```

```
    d = len(datapoints[0])

    #Limit our iterations
    Max_Iterations = 1000
    i = 0

    cluster = [0] * len(datapoints)
    prev_cluster = [-1] * len(datapoints)

    #Randomly Choose Centers for the Clusters
    cluster_centers = []
    for i in range(0,k):
        new_cluster = []
        #for i in range(0,d):
        #    new_cluster += [random.randint(0,10)]
        cluster_centers += [random.choice(datapoints)]


        #Sometimes The Random points are chosen poorly and so there ends up being empty
clusters
        #In this particular implementation we want to force K exact clusters.
        #To take this feature off, simply take away "force_recalculation" from the while conditional.
        force_recalculation = False

    while (cluster != prev_cluster) or (i > Max_Iterations) or (force_recalculation) :

        prev_cluster = list(cluster)
        force_recalculation = False
        i += 1

        #Update Point's Cluster Alligiance
        for p in range(0,len(datapoints)):
            min_dist = float("inf")

            #Check min_distance against all centers
            for c in range(0,len(cluster_centers)):

                dist = eucldist(datapoints[p],cluster_centers[c])

                if (dist < min_dist):
                    min_dist = dist
                    cluster[p] = c   # Reassign Point to new Cluster
```

```
        #Update Cluster's Position
        for k in range(0,len(cluster_centers)):
            new_center = [0] * d
            members = 0
            for p in range(0,len(datapoints)):
                if (cluster[p] == k): #If this point belongs to the cluster
                    for j in range(0,d):
                        new_center[j] += datapoints[p][j]
                    members += 1

            for j in range(0,d):
                if members != 0:
                    new_center[j] = new_center[j] / float(members)

                #This means that our initial random assignment was poorly chosen
                #Change it to a new datapoint to actually force k clusters
                else:
                    new_center = random.choice(datapoints)
                    force_recalculation = True
                    print "Forced Recalculation..."


            cluster_centers[k] = new_center


    print "======== Results ========"
    print "Clusters", cluster_centers
    print "Iterations",i
    print "Assignments", cluster


#TESTING THE PROGRAM#
if __name__ == "__main__":
    #2D - Datapoints List of n d-dimensional vectors. (For this example I already set up 2D
Tuples)
    #Feel free to change to whatever size tuples you want...
    datapoints = [(3,2),(2,2),(1,2),(0,1),(1,0),(1,1),(5,6),(7,7),(9,10),(11,13),(12,12),(12,13),(13,13)]

    k = 2 # K - Number of Clusters

    kmeans(k,datapoints)
```