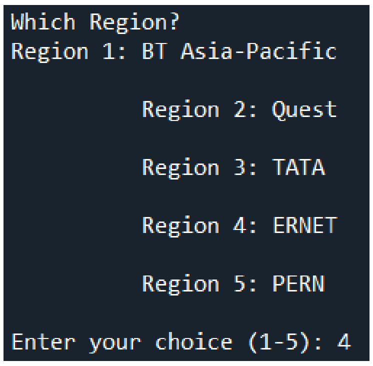
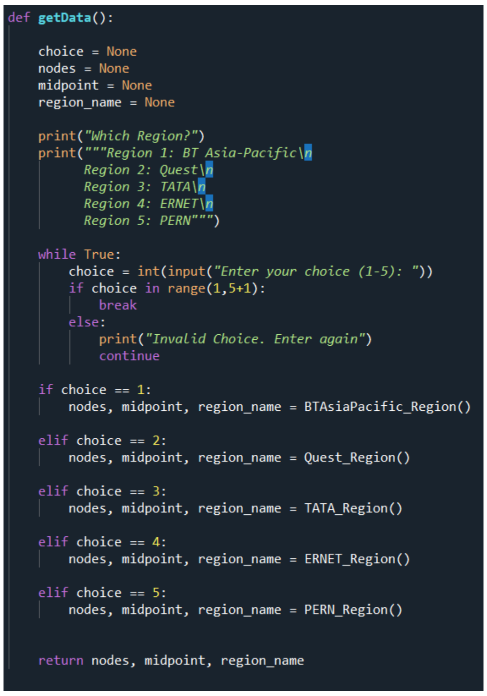
Result Analysis

Firstly, the user is asked to select the region of his choice, depending on which the details of the nodes are derived.

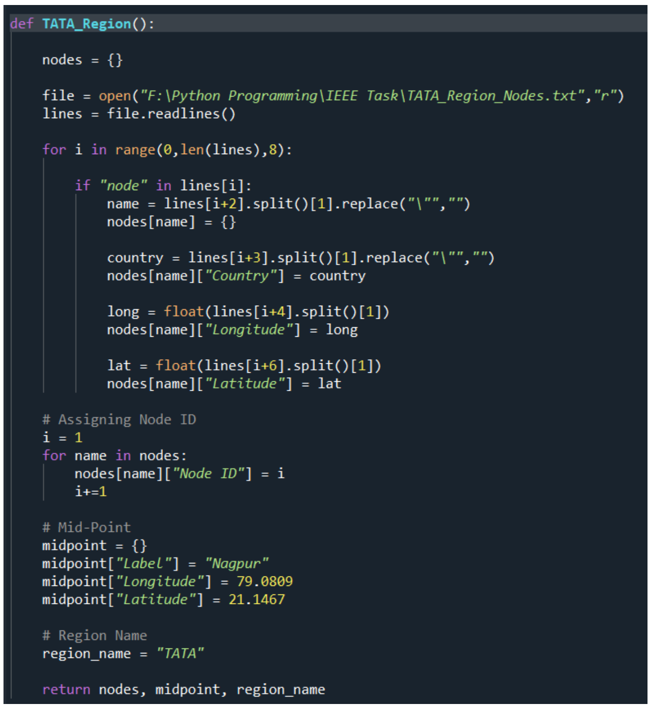


```
Which Region?
Region 1: BT Asia-Pacific

            Region 2: Quest

            Region 3: TATA

            Region 4: ERNET

            Region 5: PERN

Enter your choice (1-5): 4
```

Upon selecting the region, the details of the nodes of the selected region such as Country, Longitude, Latitude, Mid-Point and Region Name are fetched in the form of a nested dictionary from a text file derived from a GML file.

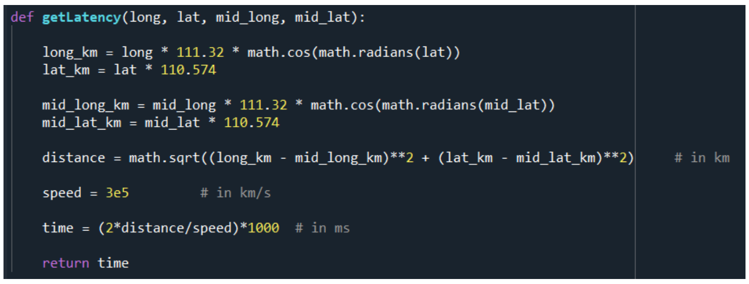


```python
def getData():

    choice = None
    nodes = None
    midpoint = None
    region_name = None

    print("Which Region?")
    print("""Region 1: BT Asia-Pacific\n
          Region 2: Quest\n
          Region 3: TATA\n
          Region 4: ERNET\n
          Region 5: PERN""")

    while True:
        choice = int(input("Enter your choice (1-5): "))
        if choice in range(1,5+1):
            break
        else:
            print("Invalid Choice. Enter again")
            continue

    if choice == 1:
        nodes, midpoint, region_name = BTAsiaPacific_Region()

    elif choice == 2:
        nodes, midpoint, region_name = Quest_Region()

    elif choice == 3:
        nodes, midpoint, region_name = TATA_Region()

    elif choice == 4:
        nodes, midpoint, region_name = ERNET_Region()

    elif choice == 5:
        nodes, midpoint, region_name = PERN_Region()


    return nodes, midpoint, region_name
```



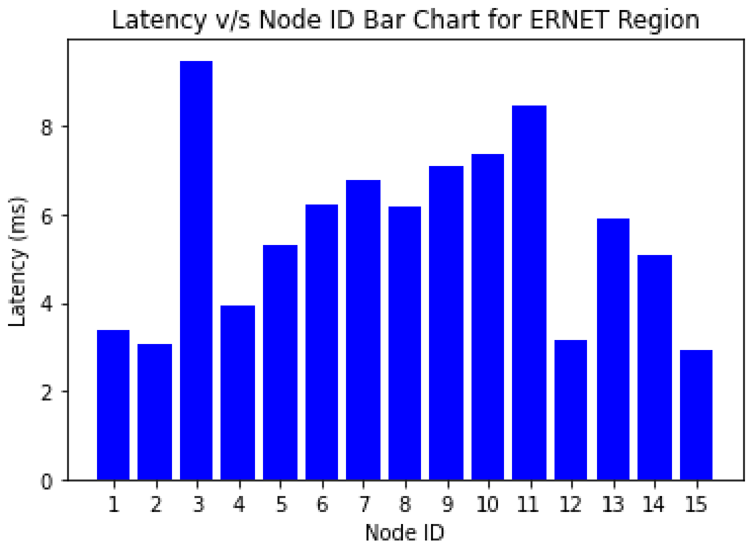
```python
def TATA_Region():

    nodes = {}

    file = open("F:\Python Programming\IEEE Task\TATA_Region_Nodes.txt","r")
    lines = file.readlines()

    for i in range(0,len(lines),8):

        if "node" in lines[i]:
            name = lines[i+2].split()[1].replace("\"","")
            nodes[name] = {}

            country = lines[i+3].split()[1].replace("\"","")
            nodes[name]["Country"] = country

            long = float(lines[i+4].split()[1])
            nodes[name]["Longitude"] = long

            lat = float(lines[i+6].split()[1])
            nodes[name]["Latitude"] = lat

    # Assigning Node ID
    i = 1
    for name in nodes:
        nodes[name]["Node ID"] = i
        i+=1

    # Mid-Point
    midpoint = {}
    midpoint["Label"] = "Nagpur"
    midpoint["Longitude"] = 79.0809
    midpoint["Latitude"] = 21.1467

    # Region Name
    region_name = "TATA"

    return nodes, midpoint, region_name
```

Latency for each node is calculated using the distance formula from the node to the midpoint. The speed of transmission is assumed to be equal to the speed of light (= 3 x 10^8 m/s) in our calculations.

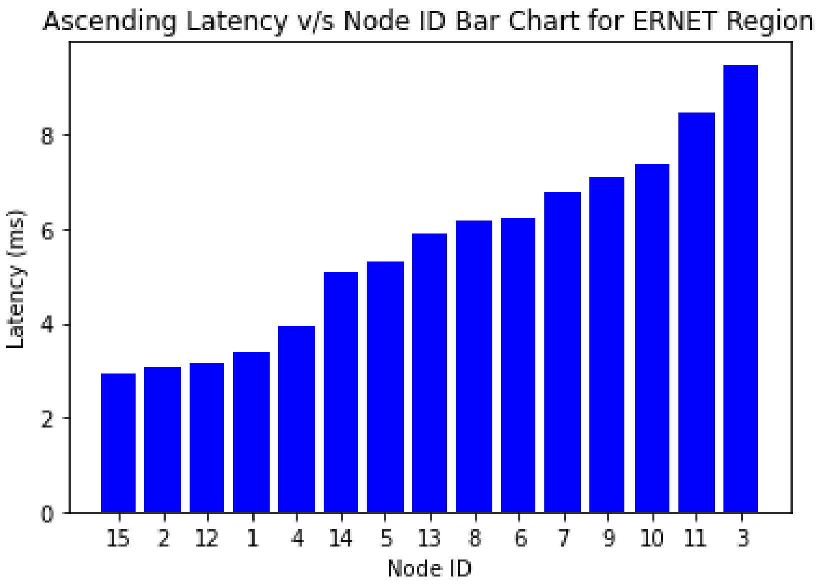


```
def getLatency(long, lat, mid_long, mid_lat):

    long_km = long * 111.32 * math.cos(math.radians(lat))
    lat_km = lat * 110.574

    mid_long_km = mid_long * 111.32 * math.cos(math.radians(mid_lat))
    mid_lat_km = mid_lat * 110.574

    distance = math.sqrt((long_km - mid_long_km)**2 + (lat_km - mid_lat_km)**2)     # in km

    speed = 3e5         # in km/s

    time = (2*distance/speed)*1000  # in ms

    return time
```

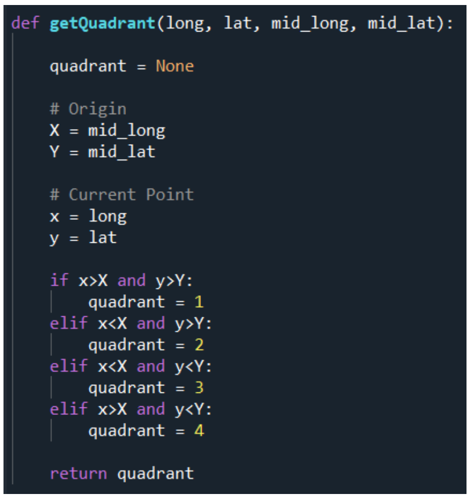
After assigning the latency parameter for each node in the region, we proceed to plot the graph for Latency v/s Node ID.



We plot another graph in the ascending order of the latencies for the nodes present in the user selected region.

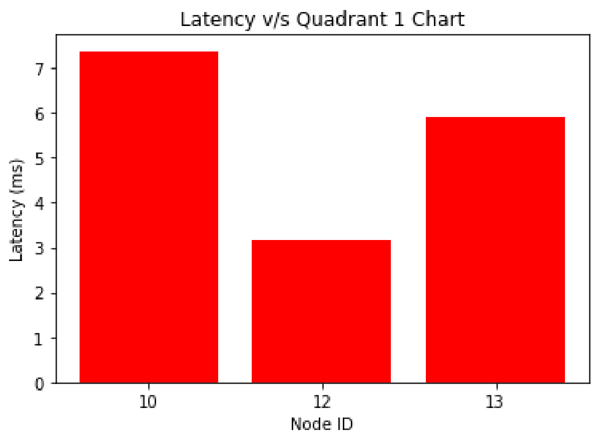


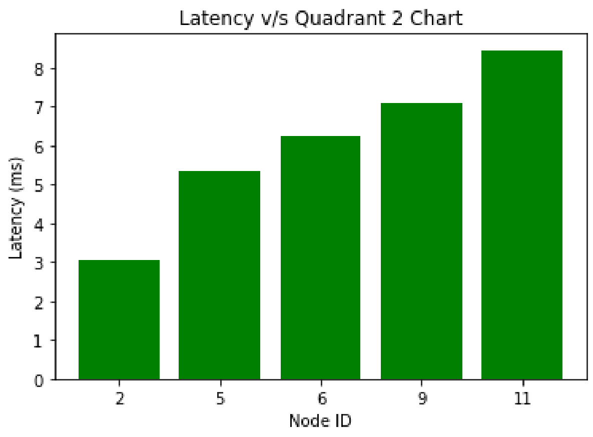
The nodes are divided into clusters based on their position with respect to the midpoint. The midpoint is assumed to be the origin and each node is assigned into a quadrant that it belongs to which is calculated using the given function:

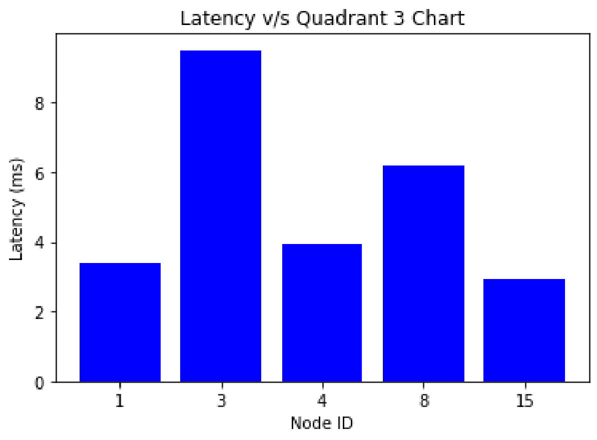


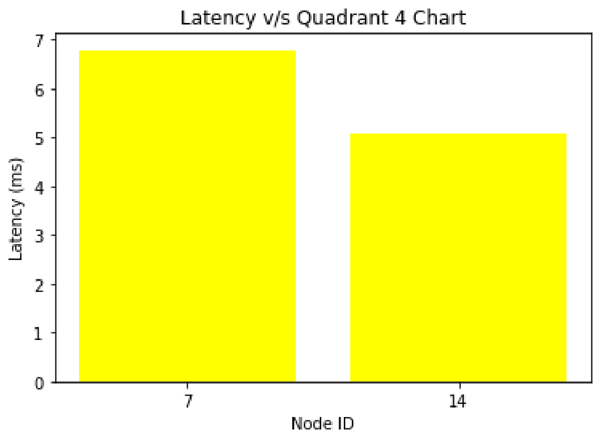
```python
def getQuadrant(long, lat, mid_long, mid_lat):

    quadrant = None

    # Origin
    X = mid_long
    Y = mid_lat

    # Current Point
    x = long
    y = lat

    if x>X and y>Y:
        quadrant = 1
    elif x<X and y>Y:
        quadrant = 2
    elif x<X and y<Y:
        quadrant = 3
    elif x>X and y<Y:
        quadrant = 4

    return quadrant
```

The bar chart for each cluster is plotted simultaneously.

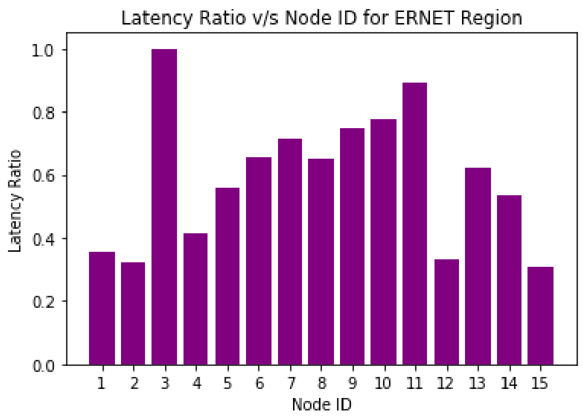




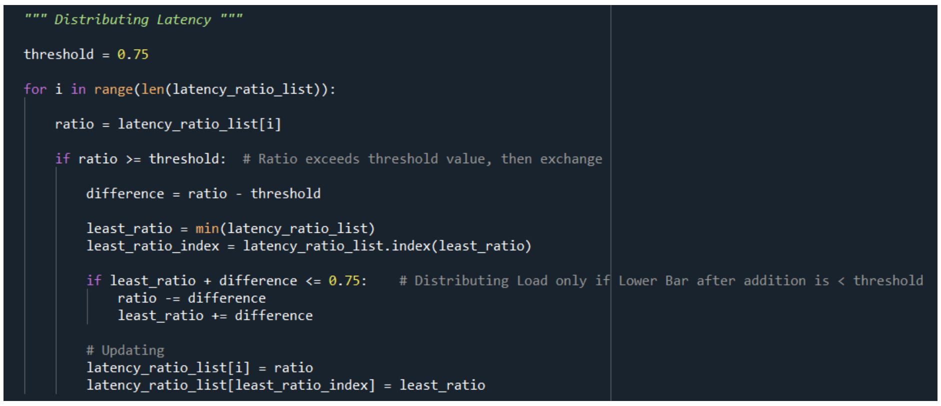




The latency ratio for each node is calculated which is the latency of the individual node divided by the maximum latency out of all the nodes present in the region. The Latency Ratio v/s Node ID graph is plotted.

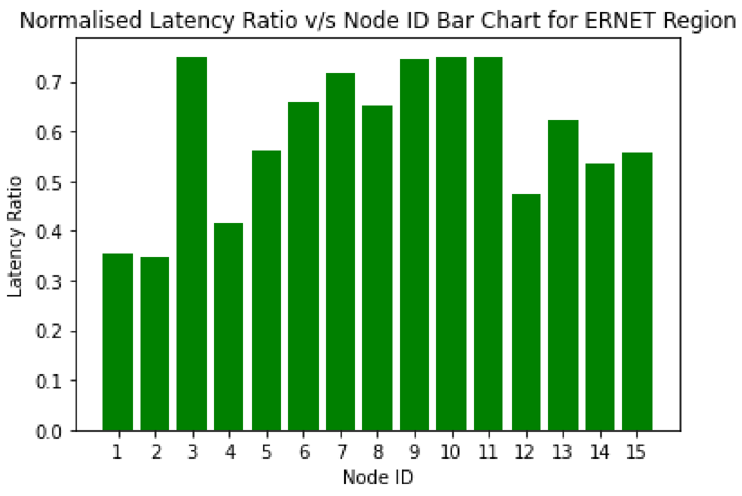


The latency is now distributed in such a way that for all the latencies crossing a certain threshold value (say 75% as in this case), the extra part is trimmed and is given to the minimum latency for the node present in the region. This is implemented by the use of the following code:

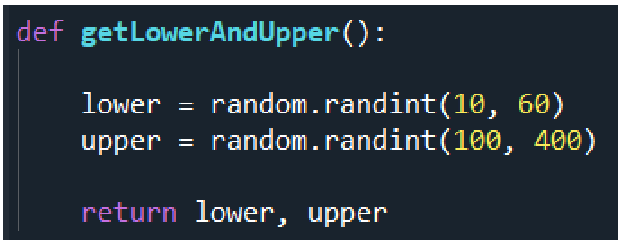


```
""" Distributing Latency """

threshold = 0.75

for i in range(len(latency_ratio_list)):

    ratio = latency_ratio_list[i]

    if ratio >= threshold:  # Ratio exceeds threshold value, then exchange

        difference = ratio - threshold

        least_ratio = min(latency_ratio_list)
        least_ratio_index = latency_ratio_list.index(least_ratio)

        if least_ratio + difference <= 0.75:    # Distributing Load only if Lower Bar after addition is < threshold
            ratio -= difference
            least_ratio += difference

        # Updating
        latency_ratio_list[i] = ratio
        latency_ratio_list[least_ratio_index] = least_ratio
```

After the normalization, the Normalized Latency Ratio v/s Node ID graph is plotted.

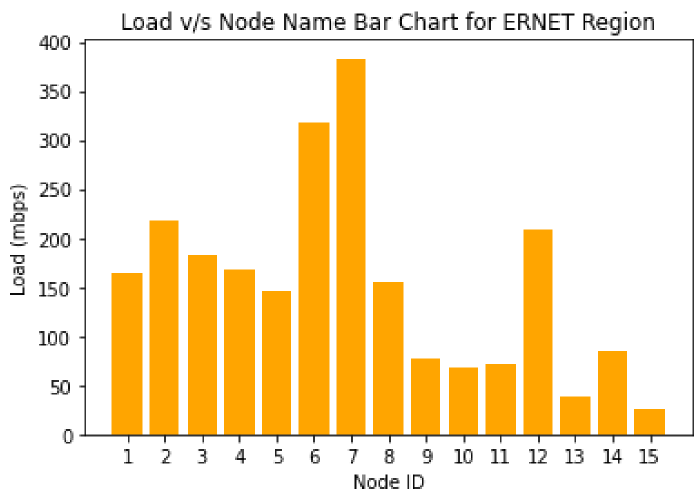


The load part of the nodes will be discussed in this section. The lower and upper limits for the load (in mbps) is randomly assigned using the function:

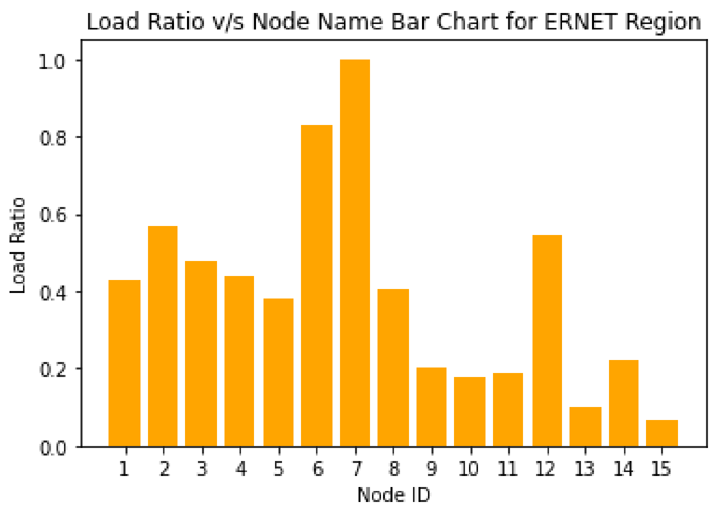


```
def getLowerAndUpper():

    lower = random.randint(10, 60)
    upper = random.randint(100, 400)

    return lower, upper
```

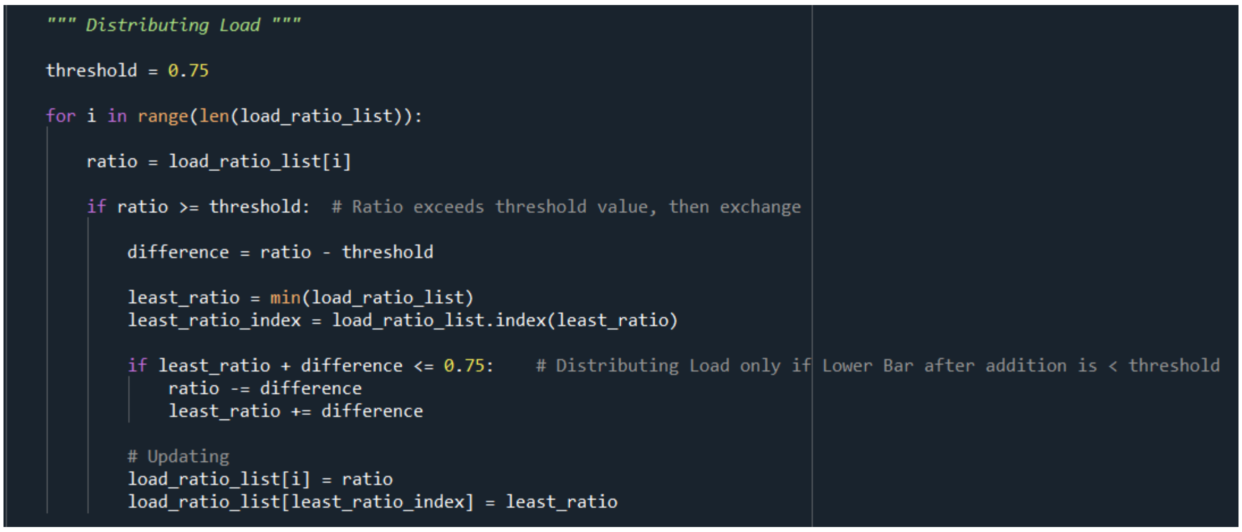
The graph for Load v/s Node ID is plotted.



The load ratio is calculated in a similar fashion as that of the latency ratio, wherein the load of each node is divided by the maximum load present in the nodes of the selected region. The Load Ratio v/s Node ID Graph is plotted.

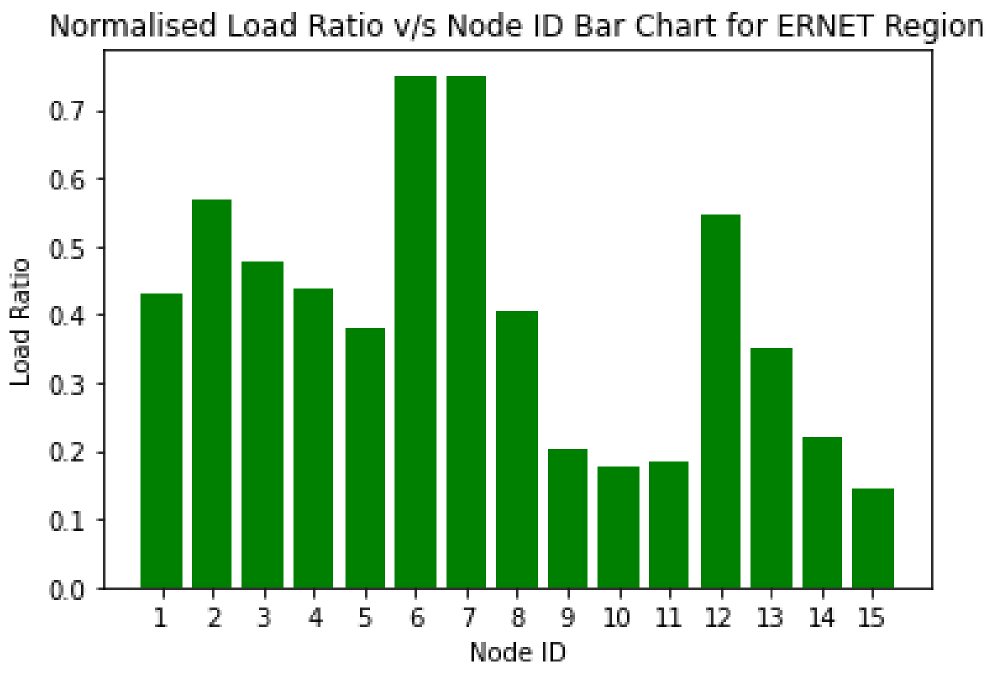


When a particular load of a load crossed a certain predefined threshold value, the extra portion which is above the threshold limit is trimmed and given to the node which has the minimum load. This is realized by the use of the following functionality:

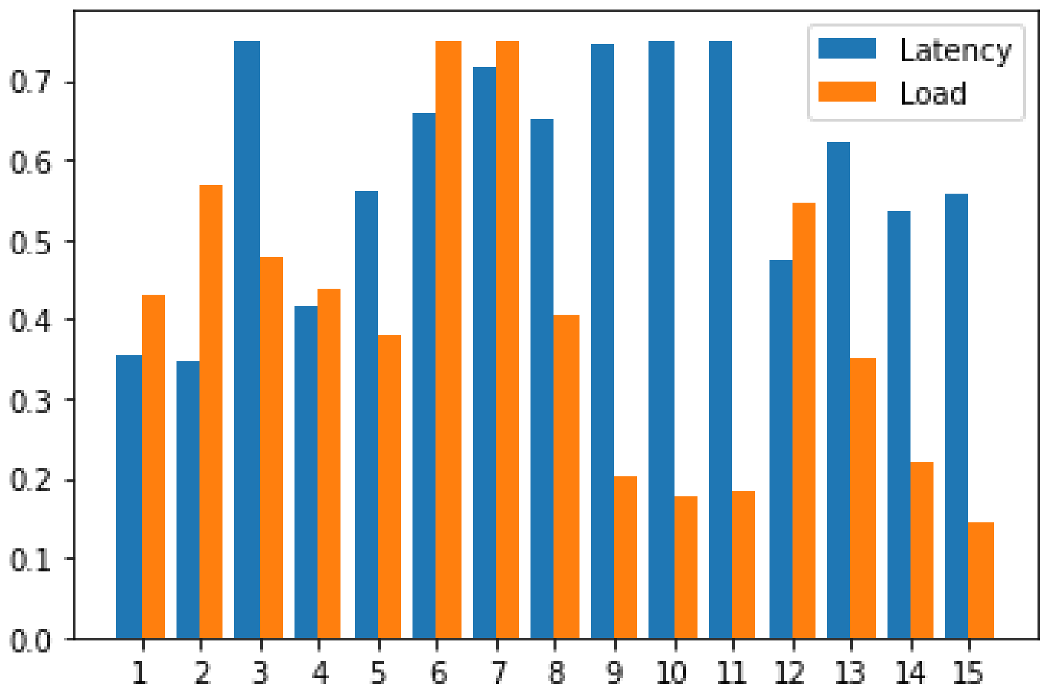


```
""" Distributing Load """

threshold = 0.75

for i in range(len(load_ratio_list)):

    ratio = load_ratio_list[i]

    if ratio >= threshold:  # Ratio exceeds threshold value, then exchange

        difference = ratio - threshold

        least_ratio = min(load_ratio_list)
        least_ratio_index = load_ratio_list.index(least_ratio)

        if least_ratio + difference <= 0.75:    # Distributing Load only if Lower Bar after addition is < threshold
            ratio -= difference
            least_ratio += difference

        # Updating
        load_ratio_list[i] = ratio
        load_ratio_list[least_ratio_index] = least_ratio
```

After this procedure, the Normalized Load Ratio v/s Node ID bar graph is plotted.



Finally, we plot the latency and load simultaneously for each node in the user selected region as a multi-bar chart.


Latency
Load