# Distributed Systems Project: Assignment : #3

Due on Friday, March 7, 2014

**Chris Blythe - 014258961**
**Farbod Faghihi Berenjegani - 014343410**
**Mohammad Bakharzy - 014083589**

March 7, 2014

# Contents

# Running the code

Our main program is included in the overlayNode.py script, but there are additional files including wrapper, configGenerator, and testOverlay. Here is a short instruction about how we ran the project on Ukko nodes. We decided to select 4 Ukko Nodes and run 256 processes on each of them. We chose 4 nodes because each Ukko node has a limit on the number of processes that an user can run on it. Our configGenerator script takes the number of nodes as an input, and creates a config file consisted of 1024 lines with three columns. The first column is the node ID, the second column is the hostname, and the third column is the port number. The hostname column of the first 256 lines is set to ukko049.hpc.cs.helsinki.fi, the hostname column of the second 256 lines is set to ukko050.hpc.helsinki.fi, the hostname column of the third 256 lines is set to ukko026.hpc.cs.helsinki.fi, and the hostname column of the last 256 lines is set to ukko052.hpc.cs.helsinki.fi. Our generated config file is also included. Then we log into our 4 selected ukko nodes listed in config.txt (ukko049,ukko50,ukko26, and ukko52). We also use another script called testOverlay.py to run a test over our overlay network. We pass the config.txt file and a number of messages we want to send over the overlay network as inputs to this script. This script randomly selects two nodes from our config.txt as the source and the destination of a message, and tell the source to send a message to the destination. For example we tried to send over 10000 messages over our overlay network by using this script. We also log into the fifth ukko node to run our testOverlay.py code. Finally we run the wrapper script on each ukko nodes included in the config file, to run 256 processes on each of them. The wrapper get the config.txt as an input. The wrapper run the processes that their hostname is equal to the localhost name of the selected node.
./wraper.sh config.txt
After all of the servers on previous four nodes started to listen, we run the test script on the fifth ukko node.
python testOverlay.py config.txt 10000
There should be a directory called testing in our working directory, so that every time that a message is received at the destination, the destination will write the hops between the source and the destination to a log file.

# Organizing the nodes(question 1)

This section is answering the first question about how we organized the nodes and why we chose this method. It also answers the forth question partially. We have chosen an approach similar to Chord protocol for this assignment[1]. The Chord protocol is based on a logical ring with positions numbered from 0 to $2^m$-1. Each node is given a logical position on the ring, which in our case is from 0 to 1023. Each node maintains a finger table as its routing table so that it can selectively route some other nodes in addition to its successor. The main logic is to obtain the routing table for each node and implement the routing algorithm. For each node in the logical ring, the routing table is calculated separately which has a fixed number of nodes to route(In case of 1024 nodes, routing table has the routing information of 10 nodes). Detailed about how to calculate which 10 nodes should be associated with a source node is presented in the next section. The main reason we chose Chord protocol over other options is the simplicity of the algorithm and having a low (H*R = 50) which is far less than 387. Another reason is that we could use our existing codes from first assignment to implement Chord protocol on top of those which saved us some time. In our context in which we do not need to consider fault tolerance issues and there are a fixed number of nodes, Chord seems to fit.

## Routing Table and Maximum Routing Table Size(question 3)

In this approach, each node maintains a routing table size of exactly $log_2$N; therefore, in our assignment with 1024 nodes, each node maintains a routing table of size 10. The algorithm for routing table construction uses the ID of the node, and constructs the routing table base on that. In each step the algorithm finds an element of the routing table. We can find the elements of the routing table by incrementally adding a power

of two to the node ID in each step. As an example, consider that the current node is the the one with ID 0, and the size of the routing table is 10. In each step we add a power of two from 0 to 10. So the routing table of node 0 contains the following nodes, $2^0, 2^1, 2^2, 2^3, 2^4, 2^5, 2^6, 2^7, 2^8, 2^9$(1,2,4,8,16,32,64,128,256,512). If the addition results in a number is bigger than 1023, we will subtract 1024 from the resulted number. And finally we find the host address and port number of each element in the routing table by searching through the config file. The routing table generation code written in python is shown here.

```
def routingTableGen():
     #list of calculated nodeID for routing table
     indexList = []

     #for each node X, there are 10 nodes that node X has-
     # -their routing info in its routing table
     for index in range(0,int(math.log(numberOfNodes,2))):
          # The index of nodes which are saved in routing-
          # -table of each individual node is calculated below
          tempIndex = nodeID + int(math.pow(2,index))
          # In case the index which is calculated is more than number-
          # -of total nodes, then a subtract is needed to find the actual index
          if tempIndex >= numberOfNodes:
               tempIndex = tempIndex - numberOfNodes
               indexList.append(tempIndex)
          else:
               indexList.append(tempIndex)
     routingTableArray = []
     # nodesList is the list of lines read from config file
     for line in nodeList:
          # indexList(calculated about) has 10 IDs(indexes) of-
          # -nodes which are supposed to be in this routing table
          indexFromFile = int(line.split()[0])
          # Find the line from config file associated-
          # -with each ID from indexList and add to routing table
          if indexFromFile in indexList:
               routingTableArray.append(line)
     return routingTableArray
# Example of one line of the routing table:
# 4 ukko049.hpc.cs.helsinki.fi 10001
```

routing table construction code

```
#Routing Agorithm - calculate next node to send message too
def routeMessage( destination ):
    #count is keeping track of the index of next hop node in the routing table
     count=0
    #ceil is the number of entries in the routing table
     ceil = int(math.log(numberOfNodes,2)-1)
    #for loop is searching to find the position of-
    #- destination between the routing table entries
     for i in range(0,ceil):
          if (destination>=int(routingTable[i].split(' ')[0])
        and int(routingTable[i+1].split(' ')[0])>destination):
               return(routingTable[count]) #returning the next hop information
          else:
```

```
                if (i==(ceil-1)):
                    return(routingTable[ceil]) #the destination ID is bigger than-
                    #-the last entry so the next hop is the last entry
                count = count + 1
    #return node WHICH IS IN LOCAL ROUTING TABLE
```

routing algorithm code

### Routing Algorithm and Hops Between Source and Destination(question 2)

The routing algorithm follows a greedy method. In each step the node tries to find the greatest node ID in its table that is less than or equal to the destination. Therefore, in our example if node 0 receives a message for destination 300, it will send the message to node 256, which is the greatest node ID in its table that is less than the destination. The worst case scenario is that one node decides to send a message to its predecessor, which would take 10 hops for the message to reach to the destination. For example if node 0 wants to send a message to node 1023 the hops are: 0 - 512 - 768 - 896 - 960 - 992 - 1008 - 1016 - 1020 - 1022 - 1023. We know that each node has a routing table of fixed size 10, and in the worst case scenario there are 10 hops between the source and the destination so the product of max routing table size multiplied by max number of hops is 100. But since the destination is chosen randomly the average hops between the source and the destination is 5, so the product of max routing table size multiplied by average hops is 50. The mentioned routing algorithm is shown here.

## Results of Testing(question 5)

As mentioned in the Routing Table construction section, each node maintains a fixed routing table size of $log_2$N, which in our case is 10. In the worst case there is $log_2$N (in our case: 10) hops between the source and the destination, and in the best case there is only 1 hop between them. Since the source and the destination are selected randomly, then the average number of hops between them is $log_2$N divided by two(in our case: 5). So the product of multiplying average number of hops for delivering the message by the maximum routing table size is 50 (H*R = 50).
To test the overlay a testing script was developed to send a defined number of messages across the network. Each message had a randomised source and destination nodes (equally weightd out of the 1024 nodes), with an integer attribute to record the number of hops experienced by the message set to zero. The messages were then sent to their randomised source nodes and routed across the network. Each hop incremented the 'hop count' attribute in the message which was recorded when the message reached its destination address. In our test, by sending 10000 messages through our overlay network the average number of hops for delivering a message is 4,9962 and the routing table size for each node is 10. So the product of H*R is 49,962 which is really close to our calculations (50). Figure 1 shows the frequency distribution of number of hops for delivering the messages. Figure 2 shows hop number and their corresponding number of messages.

## Pros and Cons(question 4)

We have adapted an approach based on the Chord distributed hash table. The key parameter for analysing the performance of the system is the number of nodes. As seen in the previous sections both average number of hops and routing table size is depended on the number of nodes.

### Advantages

- Our model has a considerably small product of H*R = 50, compared to the requirement, which is 387.

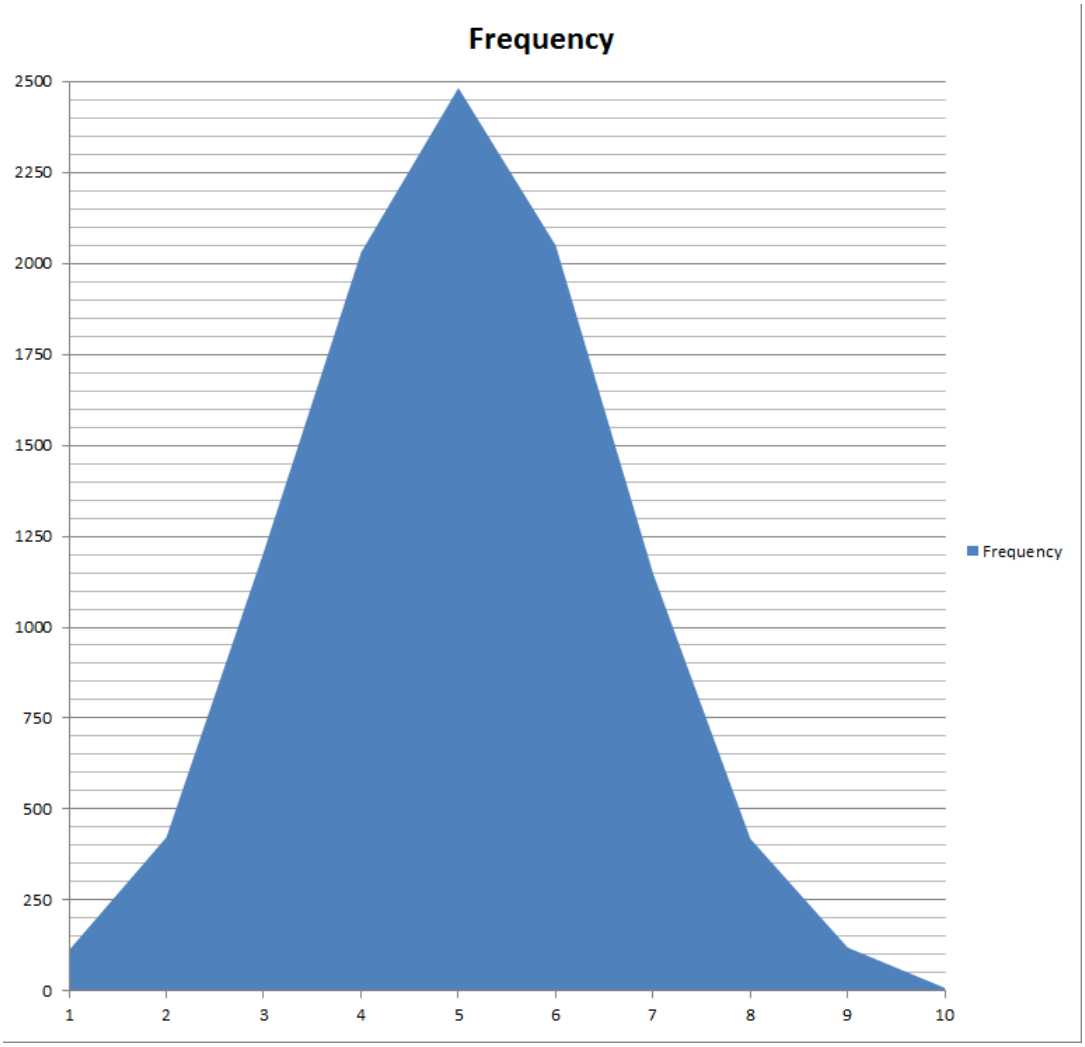


Figure 1: Frequency distribution of number of hops for delivering the message, by sending 10000 messages over the network. x-axis presents the number of hops, y-axis presents the number of messages.

| Bin | Frequency |
|---|---|
| 1 | 113 |
| 2 | 423 |
| 3 | 1207 |
| 4 | 2030 |
| 5 | 2482 |
| 6 | 2049 |
| 7 | 1148 |
| 8 | 419 |
| 9 | 120 |
| 10 | 9 |

Figure 2: Count of each hop number through our test

- The geometry of our model is based on a ring, which is easy to implement and routing.

### Disadvantages

- If one of the nodes (processes) fail, then considerable amount of messages that needed to be passed to this node in order to reach the destination will fail. Our system do not support fault tolerance.
- Our routing algorithm and routing table construction accept only powers of two as the node numbers.
- Chord distribution hash table joining and leaving approaches are similar to consistent hashing, but our approach does not support adding or removing nodes, because each position on the ring is pre-assigned to a node ID before hand, and they cannot include more than one node.
- There are other models that produce a smaller product of H*R. For example another approach based on the red-black tree gives each node a routing table of size three and an average hop number of 10, which can produce H*R=30.

## References

[1] Sasu Tarkoma, *Overlay Networks - Toward Information Networking*. CRC Press, 1st Edition, 2010.