LAB 5 REPORT

Overlay Network (Chord Implementation)

APRIL 10, 2015

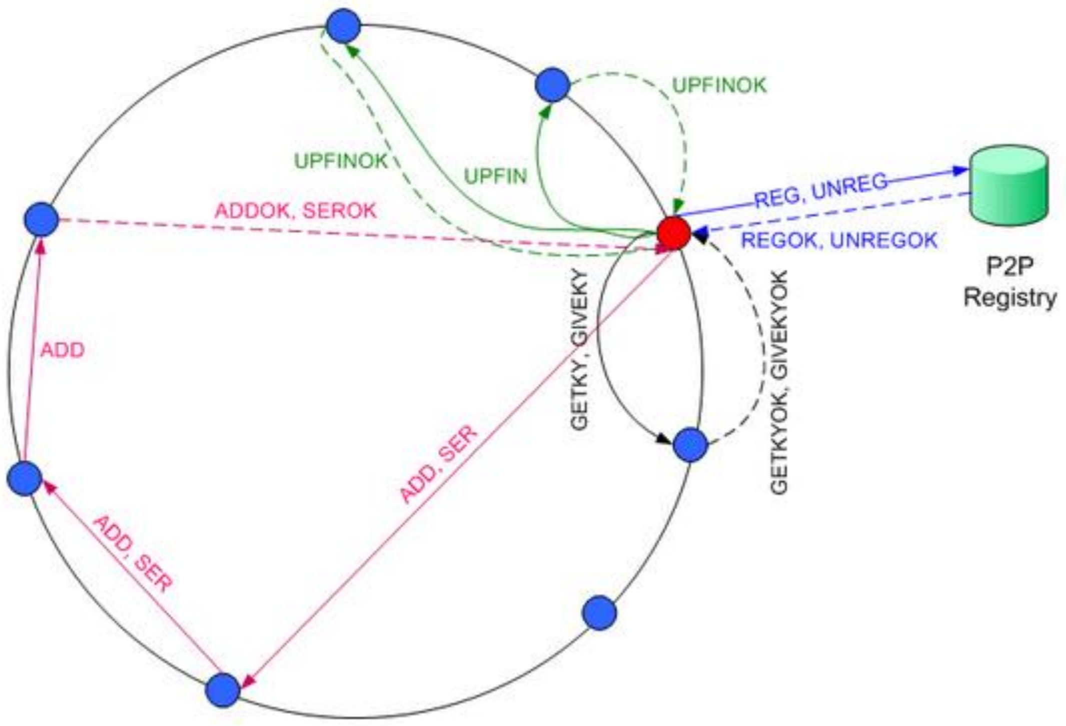
CHIRAG ANABERU BASAVARAJ
NITHIN KUMAR

Goal of the Lab:

To develop a structured-overlay-based solution that allows a set of nodes to share contents (e.g., music files) among each other. Consider a set of nodes each with a set of files that it is willing to share with other nodes. A node in the system (X) that is looking for a particular file issues a query to identify a node (Y) containing that particular file. Once the node is identified, the file can be exchanged between X and Y. Goal is to support information discovery using a structured overlay topology. To develop an efficient solution to search for contents in a large distributed system and to be able to design and develop useful structured-overlay-based applications such as simple search engines to find contents in a large Peer-to-Peer network.

Structure of the Network:

This is a structured overlay network which enables us to connect to other peers and search for the files that are present in the network. The network is connected in a manner such that each node has a predecessor and also has a successor. Basically it forms a ring. Each node in the network gets allocated with 7 to 10 files and from any node a query will be issued to search for the files present. Only content discovery has been implemented in this lab. Each node has its own finger table with all the successors and the range of owner for a particular range of key. This is done by hashing the ports and registering with the bootstrap server. Once a new node enters the network the key table and the finger table gets updated dynamically. Using this the add messages and the search queries are sent out. The algorithm we have implemented is called as the chord algorithm which is for peer-to-peer distributed hash table.



The successor is the node to the front that is in the clock wise direction to the node in consideration. The predecessor is the node behind the node in consideration i.e. in the anti-clock wise direction. Any node in the network owns a set of key space from the predecessor to itself, including its own node. This is the initial set up of the network. The protocol used for this implementation is TCP connection oriented protocol.

Implementation details:

Joining the network:

Every node which joins the network will send a register message to the Bootstrap server. The bootstrap server will send back the REGOK message along with all the nodes that are present in the network. Using this reply the node updates its finger table.

File allocation:

Every node is initialized with a set of 7 to 10 files which are stored in its file table. So when a search query comes in to that node it can respond. The file names are tokenized and hashed and are shared with under score in the file table.

Get Key:

After the node joins the network, it sends a Get key message to its successor. The successor checks its finger table and gives back all the keys which the joining node owns as of now. The joining node then adds all these keys to its finger table.

Adding keys:

After it gets its keys, the node will then check its key table and check store the files belonging to the key space it owns. If the file doesn't belong to it then it checks the finger table and forwards it to the appropriate owner of the key. This forwarded key continues to travel in network until it reaches the node which owns that key space

Search:

The implementation of search is same as add. When we search, the node iterates through the query file and take each query, check if the file is present in its own node. If it not present then it checks its finger table and routes it to the appropriate node. This continues until it finds the file and replies back with serok message

Leaving the network:

When the node wants to leave the network, it first sends an message to other nodes to update their finger table and then it gives keys to the successor and send the unregisters from bootstrap.

Results for 20 nodes:

Key table Size	Min	Max	Average	Std Deviation
Trail 1	54	350	202	87.89
Trail 2	48	287	167.5	70.32
Trail 3	60	321	190.5	96.68
Trail 4	52	346	199	72.07
Trail 5	63	334	198.5	98.44

Hops	Min	Max	Average	Std Deviation
Trail 1	0	4	2	1.00
Trail 2	1	4	2.5	1.15
Trail 3	0	4	2	0.80
Trail 4	0	4	2	1.10
Trail 5	1	4	2.5	0.82

Latency	Min	Max	Average	Std Deviation
Trail 1	0	3.7	1.85	1.12
Trail 2	0.8	4.2	2.5	0.90
Trail 3	0	2.9	1.45	1.24
Trail 4	0	3.2	1.6	0.92
Trail 5	0.8	4	2.4	1.26

Messages per node	Min	Max	Average	Std Deviation
Trail 1	46	553	299.5	125.04
Trail 2	62	483	272.5	100.03
Trail 3	56	571	313.5	137.54
Trail 4	68	527	297.5	102.53
Trail 5	54	563	308.5	140.04

Finger Table	Min	Max
Trail 1	24	24
Trail 2	24	24
Trail 3	24	24
Trail 4	24	24
Trail 5	24	24

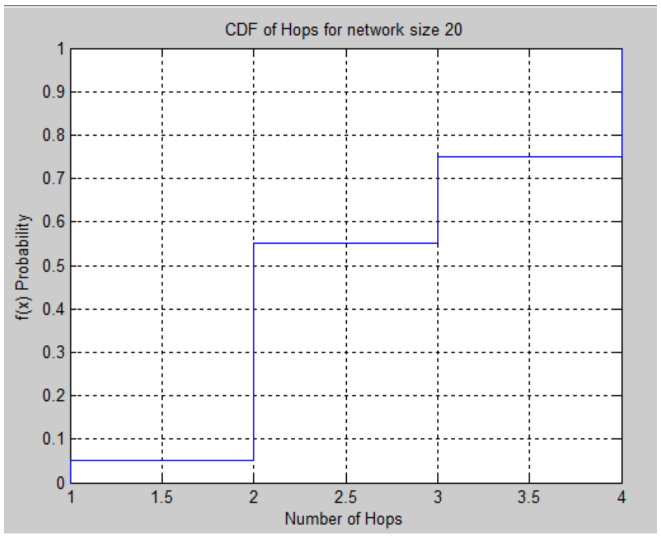
Per Node Cost = Total no of message/ Total no of nodes

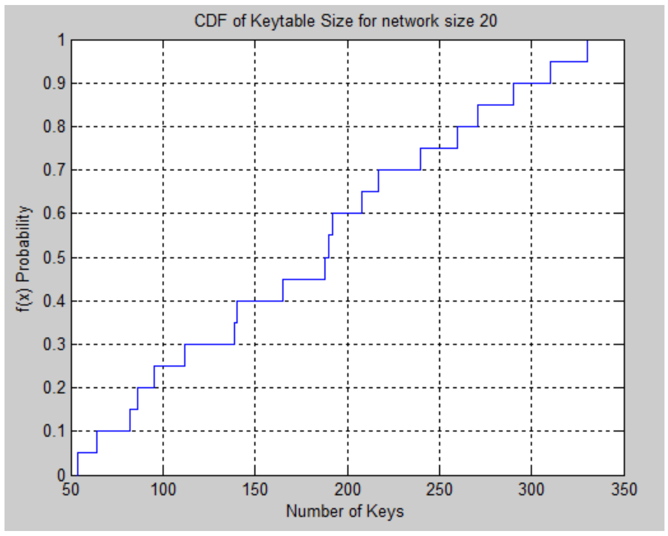
= 53.42

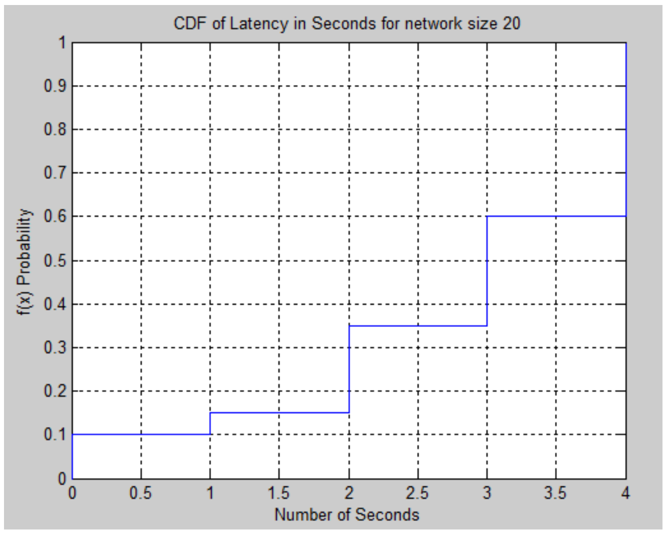
Per Query Cost = Total no of messages/ Total no of queries

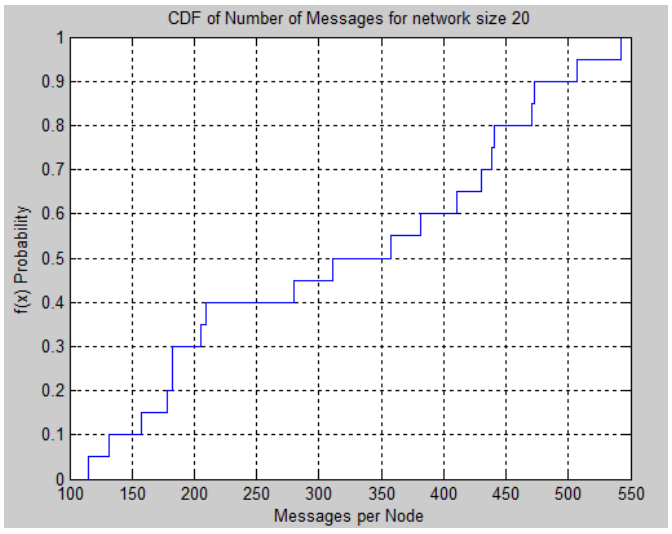
= 6.8

Graphs:









Results for 40 nodes:

Key table Size	Min	Max	Average	Standard Deviation
Trail 1	65	420	242.40	105.47
Trail 2	60	361	210.72	88.46
Trail 3	79	424	251.65	127.72
Trail 4	55	367	210.94	76.40
Trail 5	50	267	158.80	78.75

Hops	Min	Max	Average	Standard Deviation
Trail 1	0	6	3	1.74
Trail 2	1	6	3.5	1.39
Trail 3	0	6	3	1.91
Trail 4	0	6	3	1.43
Trail 5	1	6	3.5	1.95

Latency	Min	Max	Average	Standard Deviation
Trail 1	0.5	4.9	2.7	1.61
Trail 2	0.4	5.6	3	1.45
Trail 3	0.8	6	3.4	1.93
Trail 4	0.6	6.2	3.4	1.83
Trail 5	0	4.2	2.1	1.80

Messages per node	Min	Max	Average	Standard Deviation
Trail 1	64	774	419.3	175.05
Trail 2	99	773	436	160.05
Trail 3	94	959	526.68	231.07
Trail 4	101	780	440.3	151.75
Trail 5	87	912	499.77	226.87

Finger Table	Min	Max
Trail 1	24	24
Trail 2	24	24
Trail 3	24	24
Trail 4	24	24
Trail 5	24	24

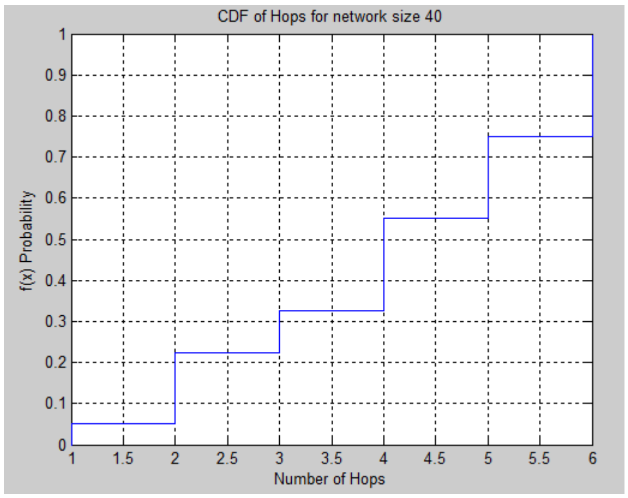
Per Node Cost = Total no of message/ Total no of nodes

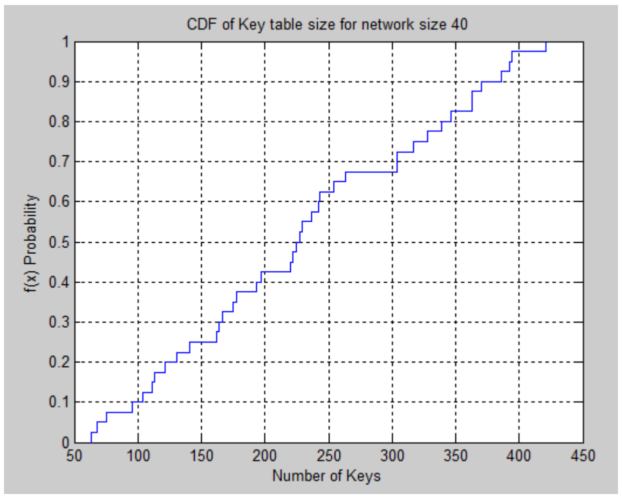
= 92.56

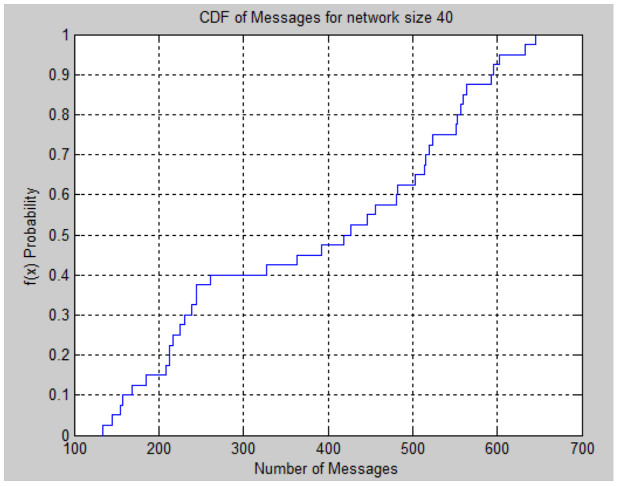
Per Query Cost = Total no of messages/ Total no of queries

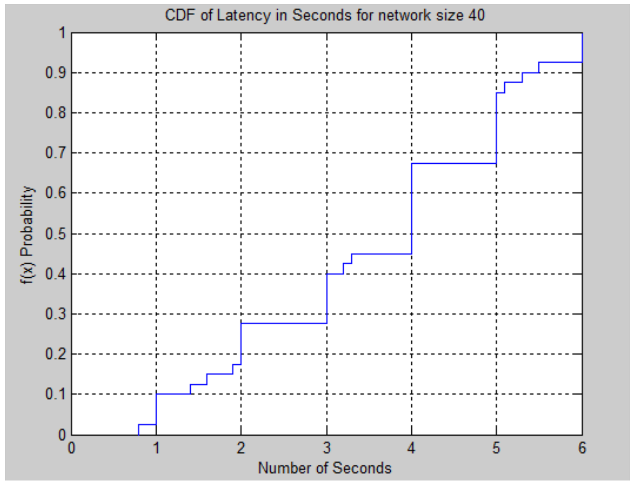
= 11.3

Graphs:









Effects of scaling the network:

When we scale the network to more no of nodes, it was observed that the latency and hops required to resolve a query generally took longer than 20 nodes. This is because the same key space is assigned to more no of nodes. This means that each node owns lesser key space as compared to 20 nodes which means there would be less no of files assigned to each node. So when we search for a file it take more hops to find the file as the files are thinly distributed in the network. Subsequently the no of messages and latency of the query would increase

Comparison with Lab 4:

In lab 4 the even though files were present in some node in network it was not guaranteed that we could find the file when we searched for it as we implemented hops limit and if it couldn't find within that limit it would just return not found. Also the query would just flood the network not checking or having a logic for routing.

Whereas in Lab 5 the nodes are connected in a structure that is in form of a ring. So when we search for a query it will mostly find within certain no if hops. If the file is present in the network it will be found as each node owns a key space and must have the pointer to the file of the key space. This is achieved as there is certain logic for the nodes to implement when they join or leave the network so that the complete key space is owned by all the nodes.

Improve Query Resolution:

We could assign the range of key space used based on the no of nodes we have to search in. If each node owns a larger part of the key space then it would store more amount of keys in its own node. This would mean that there would be more files allocated to each node. So when we search for a file from any given node it would be able to find it in less no of hops and subsequently the latency and no of messages would drastically reduce.