ECE/CS-658 Final Project

Binary Tree DHT structure for Content Distribution

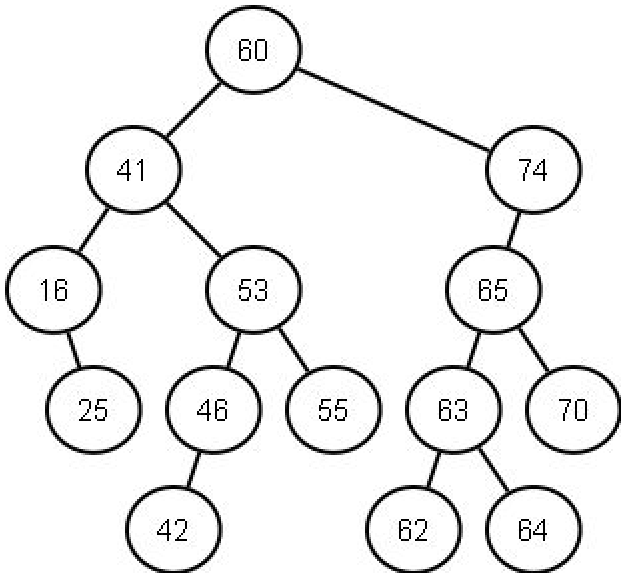
Rohit Iyer
Gandhar Vaidya

Motivation

Many widespread applications make use of DHT based P2P technology as their underlying overlay network due to their scalability, robustness, and efficiency in load balancing. Applications like file sharing, communication and live video streaming are usually in a large distributed network environment. Complex query processing is an important paradigm for DHT for an efficient and effective search in large data repositories. We wanted to implement a data structure which could tackle a multitude of problems with distributed systems such as scalability in case of Multicast and Content Distribution such as local logs in worker nodes based systems for central processing. Towards the goal of supporting complex routing in DHT-based P2P systems, this approach focuses on the usage of binary tree to build a tree-based index. Although binary tree is not the best solution and better approaches are available, we decided to use this approach as a proof of concept and documented the results.

Underlying Concept

The distributed data structure that we are implementing is a binary tree. The numbers mentioned in the nodes signify the keys for those particular nodes with the root as the top most node (In our case the first node that joins the system). Every node maintains pointers for 3 locations



- Left subtree
- Right subtree
- Root

Query generation can be done at any node in the network and the queries will propagate in one of 3 directions :

- Towards the Root
- In the left sub tree
- In the right subtree

The query remains in the network until it reaches a destination node which will be the parent of the new node in case of JOIN.
All message will contain a key representing an existing node or a resource in the system and will be propagated within the tree and will

generate a response from the node with that particular key referring to a node in its right or left sub-tree (In case of JOIN the subtree does not exist yet).
The identifier space gets halved as we move down from the root at each level of the tree. The keys will reside in a circular identifier space (ideally a power of 2). All nodes in the system will be assigned a position relative to the roots key.

Applications and Use Cases

This data structure can be used to implement a worker node based logging system wherein log entries propagate towards a centralized server. This would make narrowing down problems at certain nodes easier to identify and narrow down.
Another typical use case of this distributed data structure would be overlay based multicast. Multicasting applications send small messages to a large number of recipients at the same time without the need to establish point to point connections. The emphasis of such applications is speed over reliability. In other words, it is more important to send timely data than to ensure any specific message is actually received.

Design and Code

The most primitive class in the system is the KEY class which implements all the functions required to make routing decisions. Every node maintains pointers to its children and the root.
When a message with a key arrives it is compared to the roots key and is routed accordingly.
For statistics generation and analytics purposes we have implemented this as a multicast tree where the root is the source however the system can be easily modified to have more than one source in the system.

We decided to go with a bootstrapped approach since we wanted to find a random entry point into the network however in the real world this random entry point would be a node geographically close to the node attempting to join.

Join Flow :

1. Node contacts a bootstrap server which provides it with an entry point into the system.
2. Node contacts the entry point which routes the query within the network.
3. The query reaches the node which would be the parent.
4. The parent adds the node as a child and starts forwarding multicast packets to the node.
5. The New node is now a part of the network.

Fault Tolerance :

The system is self adjusting and implements a keep alive messaging scheme. Every parent keeps track of its children by sending periodic keep alive messages to which the child responds. This messaging scheme is tracked both at the parent and the child. If the parent notices the child is not active anymore it assumes it pruned from the network. If the child does not receive a keep alive from the parent then it assumes the parent is dead and it attempts t rejoin the tree elsewhere by contacting the root. The default timeout is set at 5 seconds and all experiments are performed accordingly. The rejoin time can be optimized by tweaking time for keep alive messages.

Due to the fault tolerant nature of this implementation we intentionally omitted a LEAVE protocol since the system is self adjusting.

Experiments :

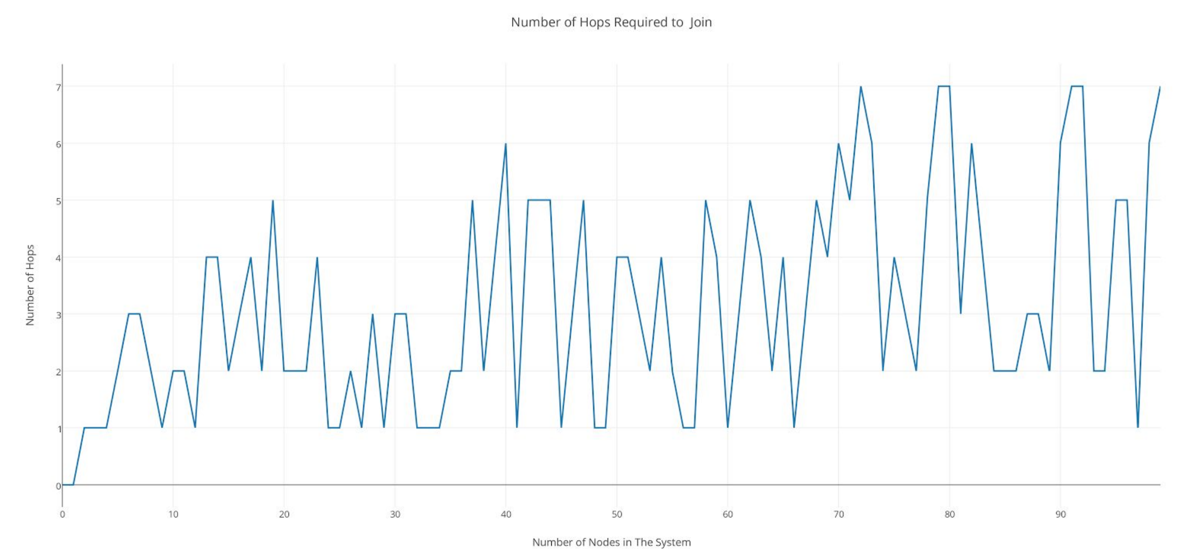
We tested our system in an identifier space of 512 nodes and added 100 nodes one by one and then made them leave one by one and documented the observations for the following properties :

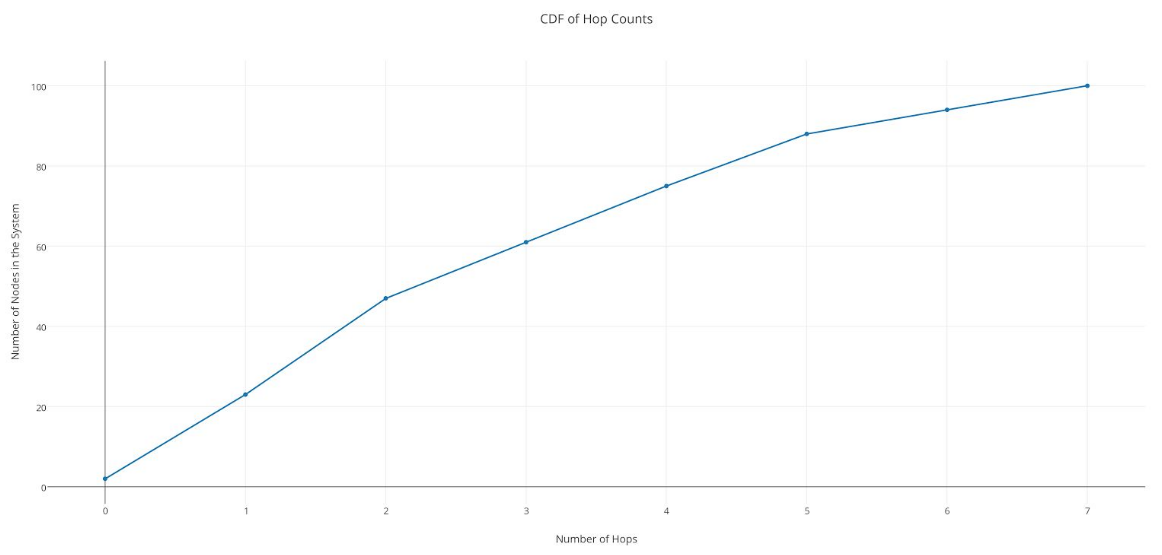
1. Hop Count
2. Latency
3. Number of messages for JOIN
4. Time required to REJOIN when parent leaves.

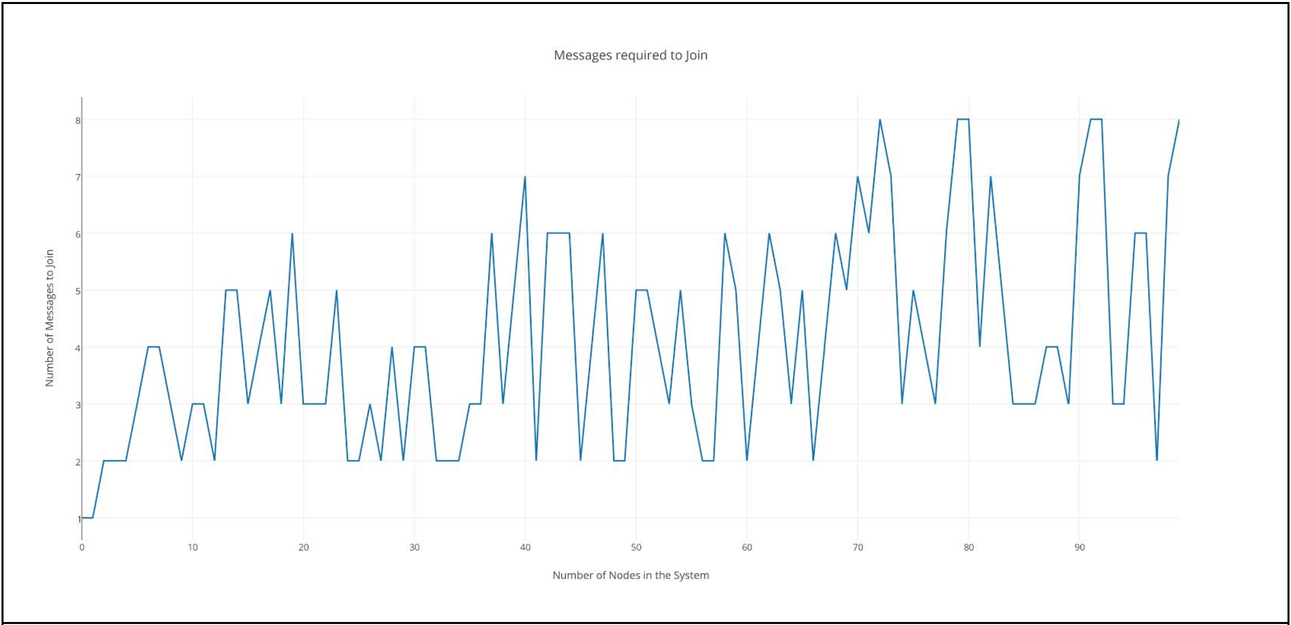
Observations and Results :

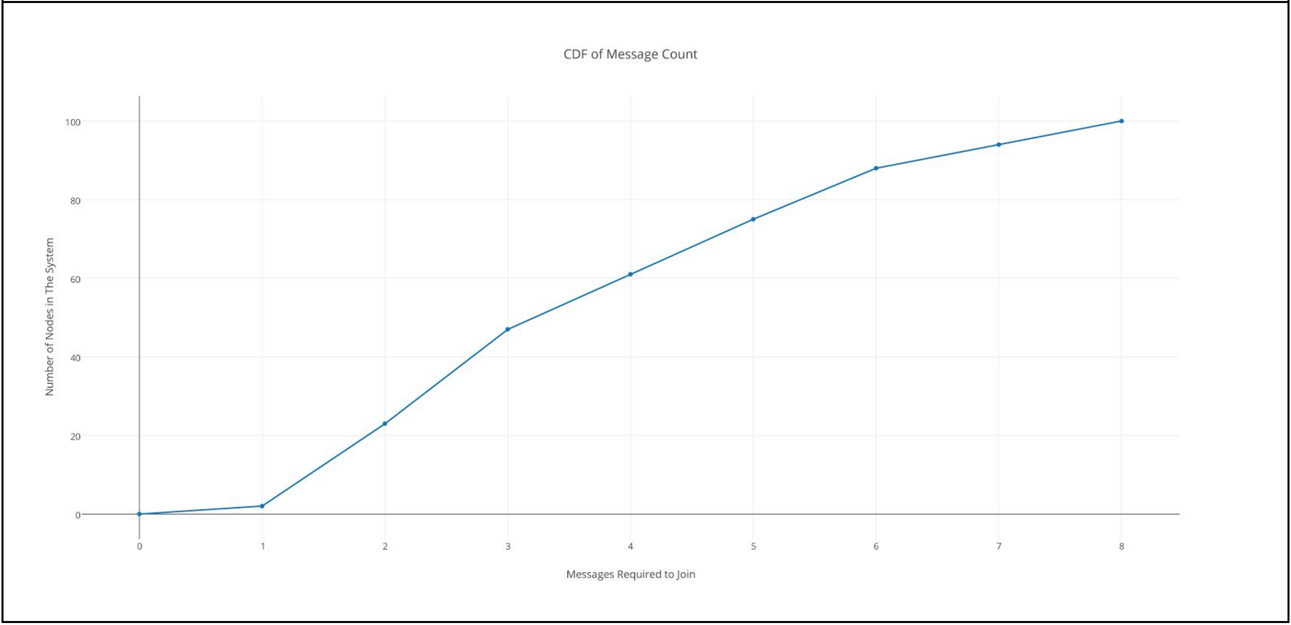
Property	Min	Max	Averaga	Std. Dev.
Hops	0.0	7.0	3.1	1.85
Latency	2ms	25ms	12.09ms	5.31
Messages	1	8	4.1	1.84
Rejoin Latency	7ms	2972ms	1475.88ms	937.25

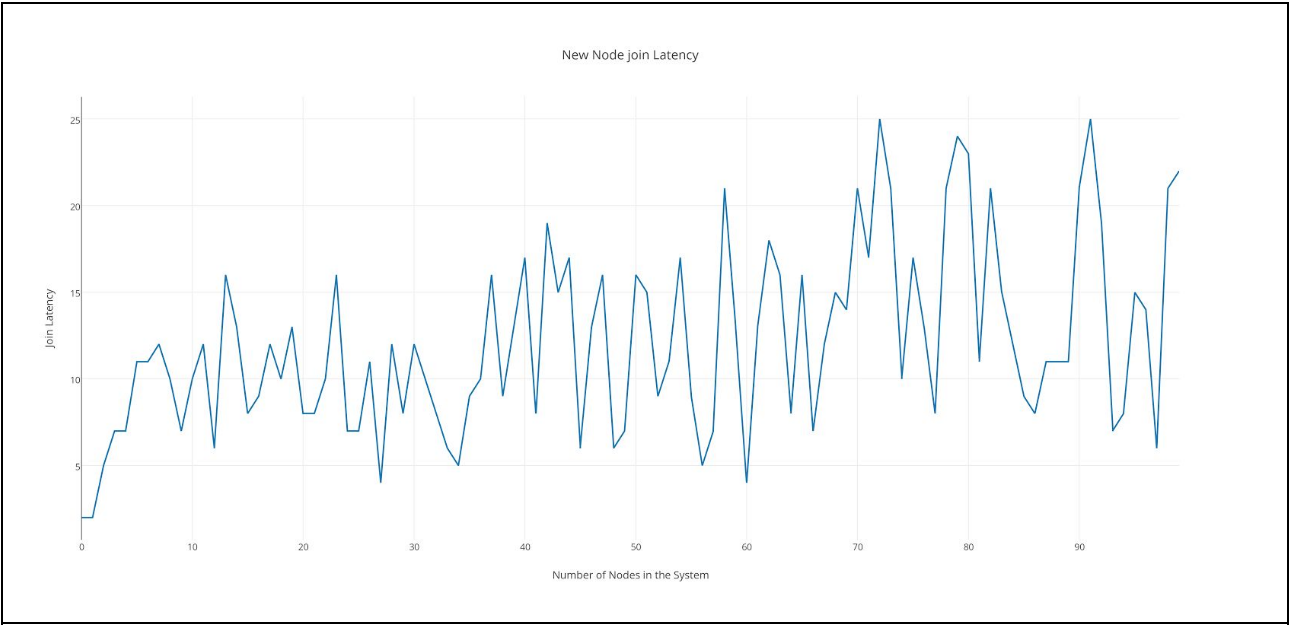
Visual Representation of Results

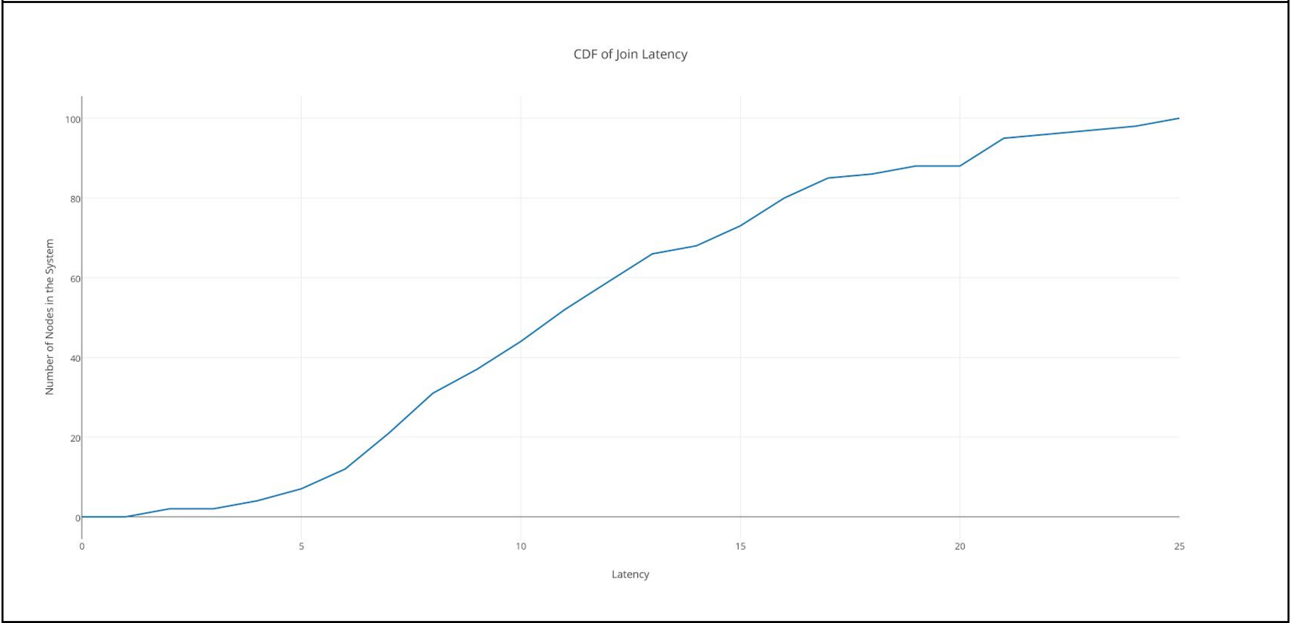


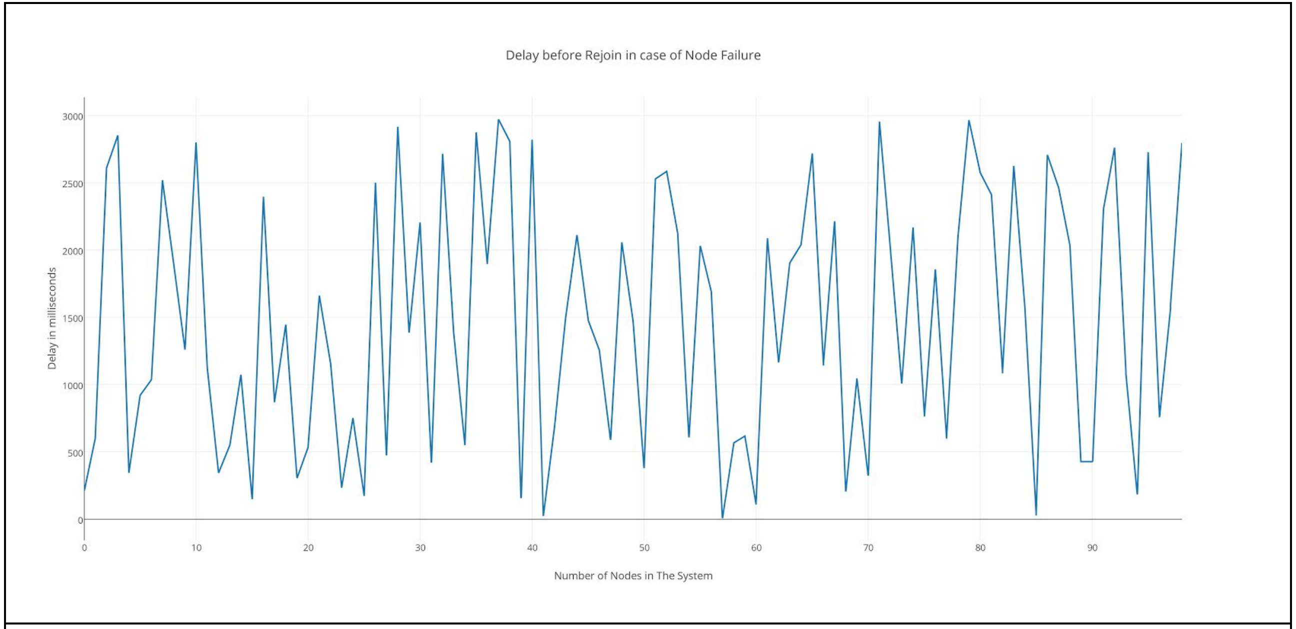


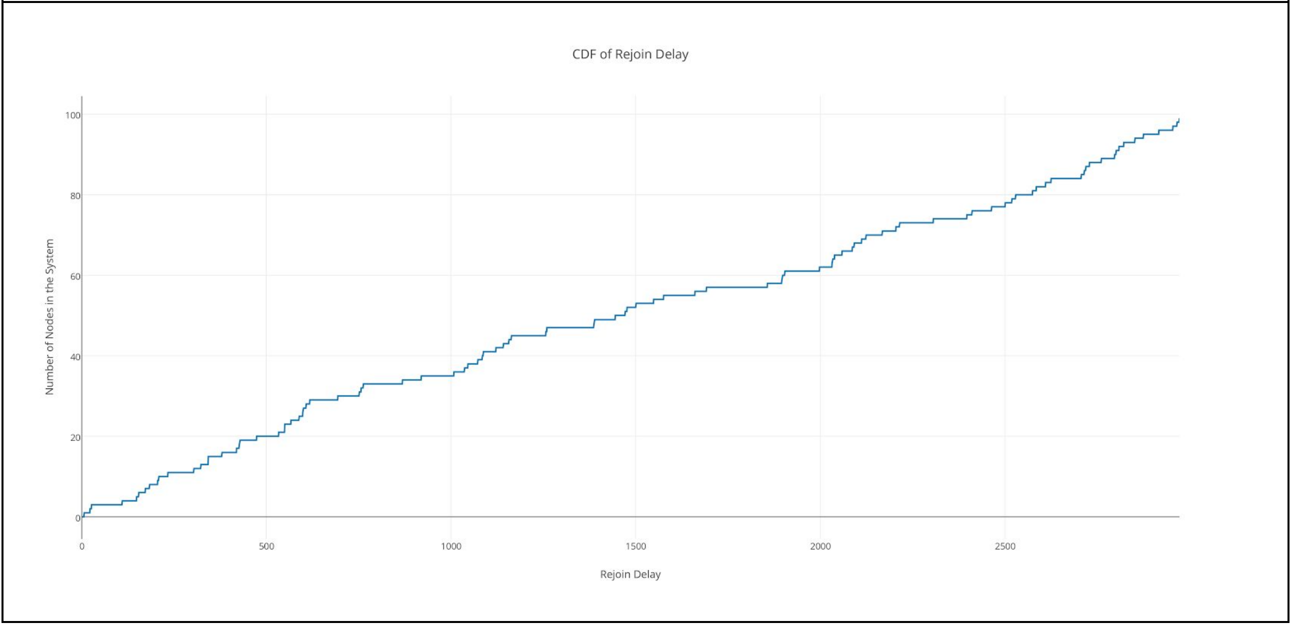

Messages required to Join
Number of Messages to Join
Number of Nodes in the System
1
2
3
4
5
6
7
8
0
10
20
30
40
50
60
70
80
90


CDF of Message Count
Number of Nodes in The System
Messages Required to Join
0
20
40
60
80
100
0
1
2
3
4
5
6
7
8


New Node join Latency
Join Latency
Number of Nodes in the System
0
5
10
15
20
25
0
10
20
30
40
50
60
70
80
90


CDF of Join Latency
Number of Nodes in the System
Latency
0
20
40
60
80
100
0
5
10
15
20
25





Conclusion & Scope of Future Work

We successfully implemented a distributed Binary tree structure and demonstrated its use case as a Multicast Network. We observed a query resolution time of O(logN). Due to self adjusting nature of this protocol we eliminated the need for a LEAVE protocol.

Future improvements in the system :

1. We can add support for splay operations within the tree.
2. For extremely large systems nodes can maintian random pointers similar to gossip protocols to improve query resolution time further.