Names-

Ekleen Kaur-94919717

COP 5615 Distributed Operating Systems Principles Project 3
Read Me

Chord Protocol

The chord protocol is the one in which when a key is given it maps it to the node. This node is responsible for storing some value for the key. Chord uses consistent hashing to do so. A chord requires O(log N) information of other nodes for efficient routing.

Note- The performance degrades whenever the information is out of date and this is important as the nodes keep on leaving or adding up.

Three features of chord that stands out from other protocols are-

- Simplicity
- Provable correctness
- Provable performance

System Model-

Load Balance, Decentralization, Scalability, Availability and Flexible naming.

Architecture-

In a chord protocol a ring is formed consisting of nodes where the first node doesn't get connected to the last node, that is node1 is connected to node2 but not to node n. Additionally, node1 has the address of the node2 but node2 will have the address of the node1 and node3 that is the previous node and the node after it. Additionally, after the creation of the ring the chord needs to be formed these are the chords of the circle. These chords are created on the basis of finger tables every node has a finger table. The finger table is the address of the connected nodes. So every node has a finger table that has the addresses of the nodes with the finger tables to whom it can travel to. When we traverse forward in a chord that time the finger table of every node has the address of the nodes in front of it. If there are 100 nodes and we traverse till the 33rd node then its finger table will consist of the address of the nodes in front of it. The creation of the finger table is created on the basis of node id+2^i where i is the position of the node. This will go on until the value of i makes the value of the finger table greater than the amount of nodes.

Key-

Data of the node that we are trying to find in the chord protocol architecture.

The following points should be noted-

- It's not necessary that every node has its own key that means it might have no data.
- Some nodes can have more than one key
- Some nodes can have only one key
- The key id will always be at the indent of the node id or at the higher precision of the node id never at a lower precision than the node id same as the key id. For example- If

we have 1000 nodes and we need to search the 95th key id then the key id will always be at that position or at a higher node but never in a node lower than that of the key id we are looking for.

Implementation-

The functions created at the actor interface for message passing are-

- **ChordMapping**- Maps all the chord in a ring architecture. It is used to create a linear ring that assigns the successor and predecessor to each node except the node1 as it doesn't have a predecessor and noden doesn't have a successor.
- **FingerTable**- Creates the finger tables using node id+2^i till the time i is less than the total amount of nodes
- **Successor**- Map data structure used to store the node addresses in the finger table
- **Predecessor**-Map data structure and it stores the node reference of the immediate previous node
- **KeyDistribution**- This function distributes the key. All keys are at its immediate successor.
- **SearchKey**-main function to implement the search which takes the keyid as an argument and returns whether the finger table of a particular node has a key or not. When we request a key search on a random node there is a check whether the keyset of the same node possesses the key or not. Upon not finding it in the key set of the randomly chosen node, the node locates the nearest possible predecessor of the keyid to be searched. To do this it starts the search in its finger table, locates the nearest possible node and then recursively searches the finger table of the chosen node for again finding the closest predecessor to the keyid. This process goes on till the predecessor is not found and the node returns true after checking the key value in its successor.

Results-

Arguments :

1 Number of Nodes

2 Number of Requests

Command: `dotnet fsi chord.fsx 1000 3`

Result Ratio : 3.439439

Command: `dotnet fsi chord.fsx 1000 2`

Result Ratio:2.324324

Command: `dotnet fsi chord.fsx 500 3`

Result Ratio:3.597997

Command: `dotnet fsi chord.fsx 100 3`

Result Ratio:4.898990

Command: `dotnet fsi chord.fsx 100 2`

Result Ratio:3.424242

Result Table-

Nodes,Requests	**Ratio**
1000,3	3.439439
1000,2	2.324324
500,3	3.597997
100,3	3.763527
100,3	4.898990
100,2	3.424242
10,1	1.111111