DOS Project 3 :
Chords Algorithm
Report

Team Members:

- Anurag Bagalwadi (UFID: 4936 9125)
- Fatema Saifee (UFID: 1508 1278)

Instructions

Expected Input

- numNodes - the number of peers to be created in the peer to peer system
- numRequests - the number of requests each peer has to make.

Report For Bonus is present in Project3-Bonus Folder

Sample

Input

mix run lib/chords.exs <numberOfNodes> <numberOfRequests>

For input numberOfNodes = 1000, numberOfRequests = 20

'mix run lib/chords.exs 1000 20'

Output

> *The average number of hops (node connections) that have to be traversed to deliver a message is 27.*

PS: Due to timer issues if in case you don't see an output and the program terminates please try once more.

Implementation

Chord is a protocol and algorithm for a peer-to-peer distributed hash table.

A distributed hash table stores key-value pairs by assigning keys to different computers (known as "nodes"); . A **Genserver Process** for each **Node** is generated, it will store the values for all the keys for which it is responsible in its state.
Chord specifies how keys are assigned to nodes, and how a node can discover the value for a given key by first locating the node responsible for that key.
A **Task** is generated for each **Lookup** function to find a keys inthe network starting from a node.

Generate random string based on the given length. It is also possible to generate certain type of randomise string using the options below:

- numNodes - the number of peers to be created in the peer to peer system
- numRequests - the number of requests each peer has to make.

When all peers performed that many requests, the program can exit.
Each peer should send a request/second.
The methods used are as follows:

1. **createNodes(**numNodes, numRequests**)**
   Creates <numNodes> Nodes, i.e. Processes. We collect all the PIDs of these processes and hash them. Finally we return a list of PIDs and their respective hashes. Arguments are as follows:
   - numNodes - the number of peers to be created in the peer to peer system
   - numRequests - the number of requests each peer has to make.

   Example
   iex> Chords.createNodes(2)
   Output
   [
   {#PID<0.122.0>, "1A2EF8ADECC2BB0CF46A7E192A015C371C9D2B4902986205D0DABDCA98D431D7"},
   {#PID<0.124.0>, "77C54B3D07894668A8B46606860276204E95BE4F3172A1A8A697D195B2358AE5"}
   ]

2. **createKeys(**numNodes**)**
   create (2 * <numNodes>) random keys using GenerateRandomStrings module.
   Arguments are as follows:
   - numNodes - the number of peers to be created in the peer to peer system

   Example
   iex> Chords.createKeys(2)
   Output
   ["4Le7C", "WKW2g", "TteAa", "kXi4L"]

3. **buildRing(**pidHashMap**)**
   Create a new Chord ring (also called identifier circle). All nodes are arranged in a ring topology, where each nodes stores the HashedPID of its successor. Main features of Chord are:
   - numNodes - the number of peers to be created in the peer to peer system
   - Load balancing via Consistent Hashing
   - Small routing tables: log n
   - Small routing delay: log n hops
   - Fast join/leave protocol (polylog time)

   The argument is as follows:
   - pidHashMap - {PID, hashedPID} list Sorted on hashedPIDs

   Example
   iex> Chords.buildRing(pidHashMap)

4. **calcfinger(**currentnode, k, m**)**
   Calculates the 256 bit value of next Finger of the <currentnode> node:
   - currentnode - The Node whose finger table we need to form
   - k - Index of the finger in the finger table
   - m - Total number of entries in each finger table

   Example
   iex> Chords.createKeys(2)
   Output
   ["4Le7C", "WKW2g", "TteAa", "kXi4L"]

5. **createFingerTables(**pidHashMap, numNodes**)**
   To avoid the linear search above, Chord implements a faster search method by requiring each node to keep a finger table containing up to m entries, recall that m is the number of bits in the hash key.
   The i^{th} entry of node n will contain successor((n+2^{i-1}),mod,2^m).
   The first entry of finger table is actually the node's immediate successor (and therefore an extra successor field is not needed).
   Every time a node wants to look up a key k, it will pass the query to the closest successor or predecessor (depending on the finger table) of k in its finger table (the "largest" one on the circle whose ID is smaller than k), until a node finds out the key is stored in its immediate successor.
   With such a finger table, the number of nodes that must be contacted to find a successor in an N-node network is O(log N).
   The argument is as follows:
   - pidHashMap - {PID, hashedPID} list Sorted on hashedPIDs
   - numNodes - the number of peers to be created in the peer to peer system

6. **assignKeysToNodes(**allKeys, pidHashMap**)**
   Assign a key to the node when hash of key is just less that hash of PID of the node.
   Key k is assigned to the first node whose key is ≥ k (called the successor node of key k)
   allKeys list in an N-node network is O(log N).
   The argument is as follows:
   - pidHashMap - {PID, hashedPID} list Sorted on hashedPIDs
   - allKeys - list of all keys to be stored in the peer-to-peer system

7. **startTransmit(**pidHashMap, allKeys, numRequests**)**
   Start the lookup task for each node in the Identity circle.
   Each node must initiate <numRequests> lookup task with a random generated key from allKeys list in an N-node network is O(log N).
   The argument is as follows:
   - pidHashMap - {PID, hashedPID} list Sorted on hashedPIDs
   - allKeys - List of all the Keys randomly generated from which a random key needs to be looked up
   - numRequests - the number of requests each peer has to make.
8. **lookup**(currentNode , keyList, totalCount, startNode)
   Search for the Key from keyList in the StartNode recursively with the help of currentNode and increment the number of hops for each lookup.
   The argument is as follows:
   - currentnode - The Node whose finger table we need to form
   - keyList - List of all the Keys randomly generated from which a random key needs to be looked up
   - totalCount - Number of Nodes * Number of Requests
   - startNode - The node who initiated the lookup