**Program Details**

**MessageDigest.getInstance("MD5"):** MD5 is used for node ID creation which is reduced to 8 character sub-string for all nodes. All IDs are created with random with **No sequential dependency** in order to test Pasty protocol to its extreme in case of Random nodes.

**Structure Details (Methods/Actors):**

1. **pastry2.scala**

* **pastry3:** main object used for starting the simulation, taking user inputs from command line arguments. Argument details:
  + **args(0):** number of Pastry Nodes to be created.
  + **args(1):** number of Messages to be passed by each Node.
* **PastrySimulator(nodeNos:Int,nosMsg:Int):** Simulator that will create and initiate all the nodes with the routing details and trigger them for message passing.
  + **node:** number of nodes
  + **nosMsg:** number of messages to be passed by each node

1. **pastryTable.scala**

* **PastryTable(ID:mutable.ArrayBuffer[String],nodes:mutable.ArrayBuffer[Actor]):** class for managing the Routing table details for each node. Routing table consist of three main section as per the paper with respect to the Node ID of the nodes — Leaf Nodes, Routing Nodes and Neighbor Nodes.
  + **ID:** contains the Array of the Routing IDs of the nodes
  + **nodes:** contains the Actor reference details of all the nodes

**both the Array are buffer array in order to add new incoming nodes.**

PastryTable contain 3 methods for Routing table creation:

* leafPredict(node:String): for Leaf Node creation for specific node
* routingPredict(node:String): for routing node creation, uses removeDups(s:String) method to generate unique digest string for each node from its node ID. This unique digest is used for deciding the Routing nodes.
* neighborPredict(node:String): random nodes to increase span of neighborhood.

1. **PastryNode.scala**

* **PastryNode(ID:String,size:Int,msgC:Int):** pastry node class defines the structure of the nodes.
  + **ID:** it’s the unique MD5 hash ID of each node which is used for identification
  + **size:** it is the limit for random calling to avoid exception.
  + **msgC:** number of message to be passed by each node using the pastry protocol

1. **CaseClasses.scala:**  contain all the Case classes shared between all the classes for message passing.

**Working details:**

* PastrySimulator is created in porject3.scala and the command line arguments regarding number of Pastry nodes and numbers of messages to be passed by each nodes is added while creation of the node.
* PastrySimulator creates the Nodes with Hash ID using **MD5 hash function.**
* PastrySimulator sends the ID and Actor reference details to PastryTable class which stores the details in ID and nodes ArrayBuffer and waits for the nodes message for **Routing table creation.**
* PastrySimulator waits for the main method to trigger the simulation, once the simulation is triggered from main method PastrySimulator send Reference of PastryTable to all Pastry Nodes.
* On receiving Pastry Table details, Pastry Nodes asynchronously calls the Pastry Table to fetch the routing table details.
* Once all nodes receive there routing table details the nodes are triggered for message passing using the Pastry Protocol methodology. Nodes are selected in random and **Destination node selected in Random in all cases.**
* Each node sends the Average Hope count to Boss PastrySimulator through ExceptionHandle and PastrySimulator keep track of the average number of hop details send by each node.
* Once PastrySimulator receive details from all nodes, it calculates **Average hop count per node** and print details.

**Testing with Number of Node:**

|  |  |  |
| --- | --- | --- |
| No. of Nodes | Time taken(msec) | Avg. Hop Count |
| 100 | 210 | 2-3 |
| 500 | 325 | 5-7 |
| 5000 | 662704 | 9-12 |

**Important observations:**

1. **Loop issues:** some time while message passing using Pastry protocol as the Routing table are generated separately for each node, it causes loop formation because of most same references. Because of same references some time the message is passed in a loop while selection of best possible option as next hop in routing.
2. **HashMap avoidance:** usage of Hash Map should be avoided as we need direct reference only once while routing when the routing table contain the needed destination ID. For all other cases full routing table search is done to get the most suitable next hop ID.
3. **Insufficient Leaf and Routing case:** in case of random node where node ID are not generated in sequential dependency getting sufficient number of Leaf and Routing nodes is not possible. It case insufficient node details for routing increasing the numbers of hop.
4. **Same Hash:** two different nodes can create same hash but vice-versa is not possible.
5. **Notification on Failure to Base node:** as next hop is not able to track the path of the base node in case of notification of failure in the standard protocol structure.

**Experiments:**

1. To overcome **Loop Issue,** set of random nodes (Neighbor Nodes) are added causing increase in span of the routing table. It helps in decreasing number of hops and access to different nodes apart from regular close nodes.
2. ArrayBuffer are used in place of **HashMap** in order to avoid complex time consuming full table search while selection of best possible next hop.
3. **Same Hash** issue is avoided by adding the ID of the node along with the Hash Code so with same hash code we can distinguish the nodes with ID and Hash Code combination.
4. **Node Failure** is controlled by notification of any node failure to Bose node. In case of permanent failure Bose node send the message of failure to all other nodes to avoid false reference of the failure nodes.