CSE 535
Asynchronous Systems
Final Report

Improving and Evaluating Pastry Implementations In DistAlgo

Team 30 -
Akhil Bhutani (110898687)
Nikhil Navadiya (112046312)
Vivek Kumar Sah (112071655)

Supervised by:
Prof. Annie Liu

1. Problem and Plan

1.1. Problem Description

Background: Pastry is a scalable, distributed object location and routing substrate for wide-area peer-to-peer applications [1]. Pastry performs application-level routing and object location in a potentially very large overlay network connected via internet and can be used to support a variety of peer-to-peer applications, including global data storage, data sharing, group communication and naming. And with Pastry's ability to realize the scalability and fault tolerance it is widely suited for peer-to-peer application.

Problem: Our problem statement was to 'Improve and Evaluate Pastry Implementations'. For this current project, we are concentrating on the aspect to **Program Better**, i.e. to reimplement or improve an existing program using the language used instead of a new language, with strong justification for the reimplementation or improvement. With this goal in mind, we searched for implementations of pastry in various languages on the web and were able to find only one good implementation in python. Another best implementation was provided by Professor Liu in DistAlgo. But after studying both the implementation carefully, we found that they were not following paper [1] closely. Hence we implemented pastry in DistAlgo closely following the paper and overcoming the errors in previous implementations, also performing a thorough evaluation of our implementation. We have also outlined in section 3.3.1 and 3.3.2, the possible errors in current implementations. For measuring the performance metrics we used average number of routing hops in case of with and without node failures, probabilistic evaluation for number of hops, average initialization time, average node join time and average lookup time as metrics. Apart from these controller was used to measure and report total user time, total system time, total process time, total memory and total wall clock time.

Input / Output

Input	Output
Pastry implementations in various languages including DistAlgo and as described in paper [1]. We will have a PastryNode class with the following parameters :- ● routingTable ● leafSet ● neighborhoodSet	Best Pastry Implementation in DistAlgo with improvised version of paper [1]. This implementation will closely follow all the different aspects of the algorithm presented in paper [1]. Some of the correctness / performance test will be carried out as part of the implementation to make sure that the

• ipAddress • Nodeid • Coordinates • 'b', by default 4 • keySize, by default 128 The PastryNode class represents the attributes for every pastry node process mentioned in the paper.	paper is closely followed. Some of the correctness metrics include: • Reachability - If src and dst node is available, then the message should be routed among them. If this happens for a fixed number of trials, we can safely assume correctness property holds. • Hop count - Expected number of hop count in a Pastry network of N nodes should be $\log_{2^b}N$. Some of the performance metrics include: • Routing performance - Average number of hops with / without failures. • Probabilistic evaluation for number of Hops. • Average initialization time for Pastry network. • Average node join time for Pastry network. • Average lookup time with / without failures.

1.2. State of Art

While searching on the web we found some of the best implementations:

- *https://github.com/kapilthakkar72/Pastry-and-Chord-DHT* [2][Python]: This was one of the best implementations available on the web. It followed a sequential approach to pastry but was written very clearly and was quite easy to understand.

- *https://github.com/unicomputing/pastry-distalgo-2012-Ken-Koch[3]* [DistAlgo]: This best implementation was shared by professor Liu. This was the only implementation available in DistAlgo. It has been well written with good documentation that made code easier to understand.

We also found other implementations in different languages but they were either poorly written or had little to no documentation making it difficult to understand. Some of them are:

- *https://github.com/ctebbe/P2P-Pastry-Implementation* [5][Java]
- *https://github.com/hamersaw/BasicPastry/tree/master/src/main/java/com/hamersaw/basic_pastry* [6][Java]
- *https://github.com/ngrj93/Pastry* [7][C++]

Besides these implementations of pastry we also came across some interesting real-time applications of pastry:

- **SCRIBE [8]**: SCRIBE is a generic, scalable and efficient group communication and event notification system. It provides application level multicast and anycast. Scribe is efficient, self-organizing, flexible, highly scalable and supports highly dynamic groups. It is built on top of Pastry, a generic, scalable, self-organizing substrate for peer-to-peer applications.
- **PAST [9]**:PAST is a large-scale, peer-to-peer archival storage utility that provides scalability, availability, security and cooperative resource sharing. Files in PAST are immutable and can be shared at the discretion of their owner. PAST is built on top of Pastry, a generic, scalable and efficient substrate for peer-to-peer applications.
- **POST [10]**: POST is a generic messaging infrastructure that is built upon Pastry. It is being used to support services like secure email (ePOST), secure instant messaging (imPOST), and collaborative applications like shared calendars, notes and whiteboards without the need for dedicated servers.

1.3. Tasks

Following were the main tasks that were performed:

- Understand Pastry algorithm as presented in the paper[1].
- Understand Pastry from two best implementations mentioned in section 1.2.
- Implement an optimized Pastry algorithm in DistAlgo, closely following the paper [1].
- Verify correctness, measure performance and evaluate our implementation based on metrics like node initialization time, time taken to route join request, time taken for lookup etc.

Following were the sub tasks that were performed:

- Read and understand Pastry implementation from paper[1]
  [Akhil Bhutani, Nikhil Navadiya, Vivek Kumar Sah]
- Understand various pastry implementations:
  - Pastry implementation in Java[5][6] [Nikhil Navadiya]

    - Pastry implementation in Python[11][2] [Vivek Kumar Sah]
    - Pastry implementation in C++[7][12] [Akhil Bhutani]
    - Pastry implementation in DistAlgo [3] [Akhil Bhutani, Nikhil Navadiya, Vivek Kumar Sah]
- Implemented node arrival, core routing algorithm of pastry [Nikhil Navadiya, Akhil Bhutani]
- Implemented node delete [Vivek Kumar Sah]
- Implemented various performance metrics as mentioned in paper[1]. [Akhil Bhutani, Nikhil Navadiya, Vivek Kumar Sah]

1.4. Project Plan

Below is the project weekly progress plan:

- **Week 1:**
    - Studied Pastry as mentioned in paper[1]
    - Understood various pastry implementations by comparing with various aspects of the paper.
- **Week 2:**
    - Completed design document for various modules of pastry.
    - Started pastry implementation in DistAlgo based on planned design document
- **Week 3:**
    - Completed pastry implementation in DistAlgo
    - Performed thorough evaluation of code and checked for possible bugs
    - Started implementation of various evaluation metrics as mentioned in paper [1]
- **Week 4:**
    - Completed implementation of evaluation metrics and compared generated results with results presented in paper.
    - Performed any further code optimizations
    - Performed code reformatting and added necessary comments
    - Generated final performance matrix
    - Preparing for final presentation and report.

1.5. Motivation

The primary motivation of this project is to understand the Pastry [1] and implement an optimized version for the same in DistAlgo. The motive behind implementing Pastry in DistAlgo from scratch is to create an implementation which closely follows all the aspects of pastry as mentioned in [1]. The current implementations do not closely follow and also deviate from the actual paper [1] in one way or the other as mentioned in 3.3.1 and 3.3.2. We also intend to

compare our implementation using various comparison and performance metrics described in the paper and test performance using metrics like running times, average routing hops etc.

We chose DistAlgo as programming language as all other implementations are coded were in languages like C++, Java, Python etc. Also, as DistAlgo is a language for distributed systems, we feel it will do justice to implement the algorithm presented in the paper [1] in DistAlgo [4]. Also, based on perusing through current implementations, we don't feel that any implementation closely follows the paper and hence, we feel there is great scope for improvement.

2. Design

2.1. Overall Architecture

2.1.1. Class diagram

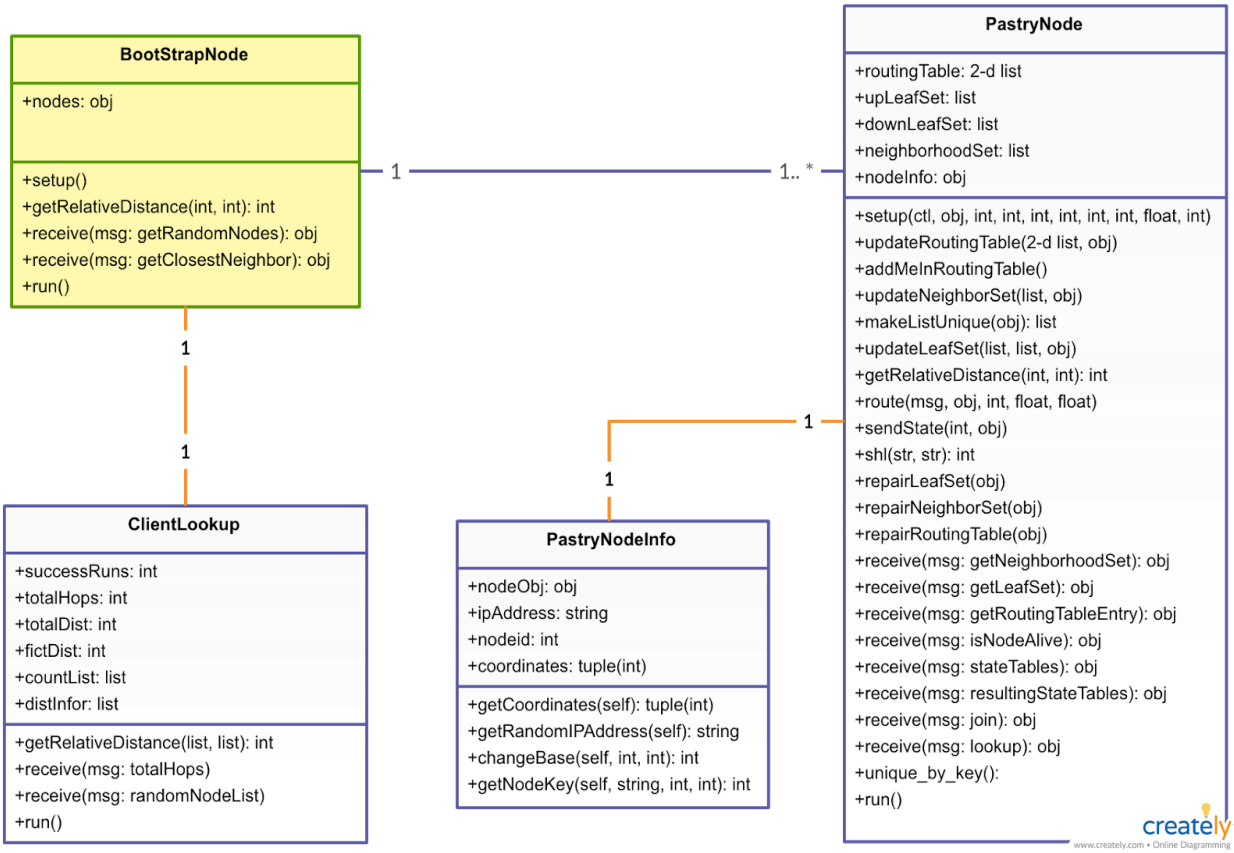

Fig. 1. Class Diagram Representing all Classes

The class diagram presented in figure 1 represents the classes defined in our program. This also depicts the relationship with various classes and member functions of each classes along with global variables accessible to each method.

2.1.2 Sequence diagram for Client Lookup

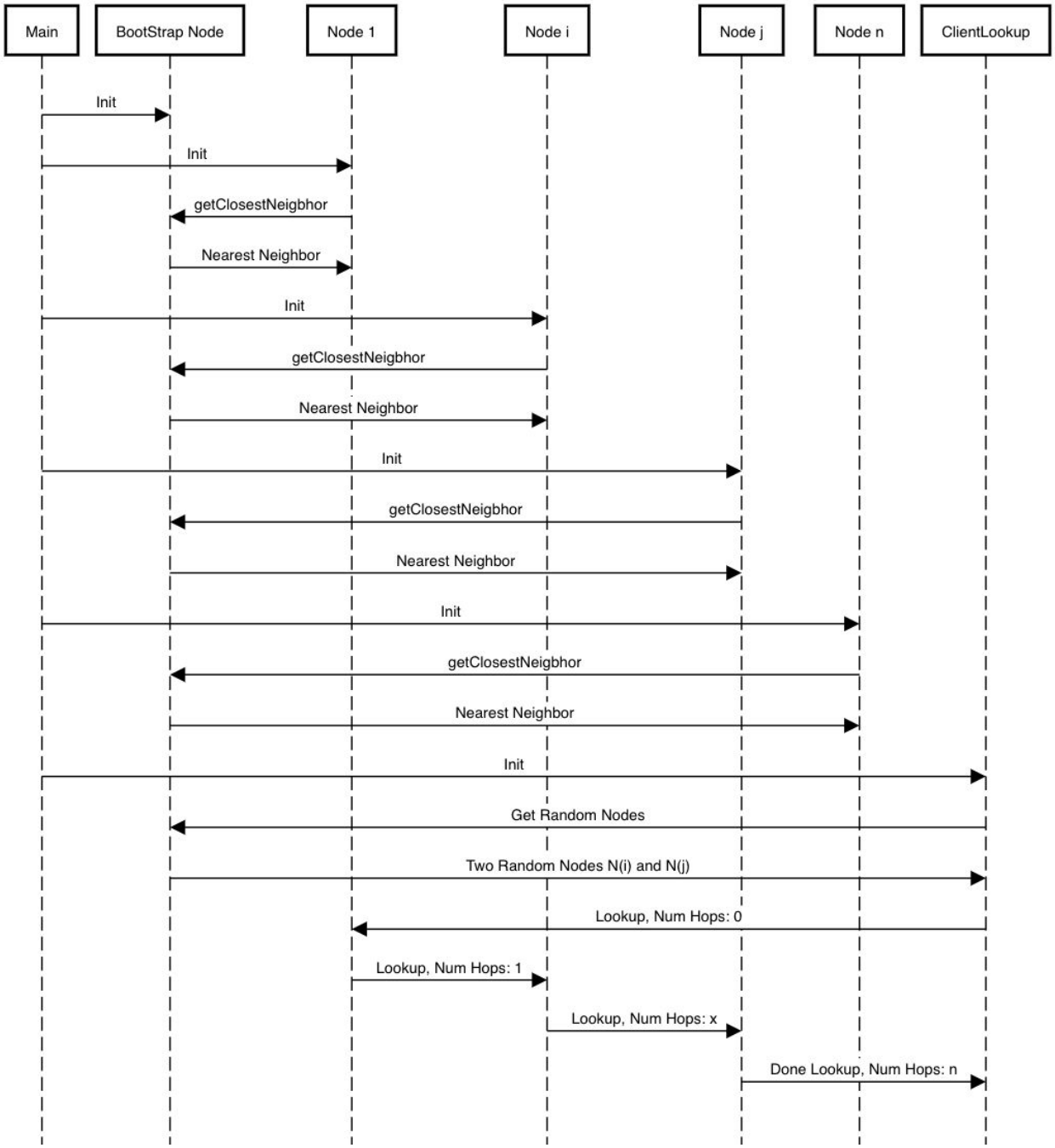

Fig. 2. Client Lookup Sequence Diagram

2.2. FlowChart

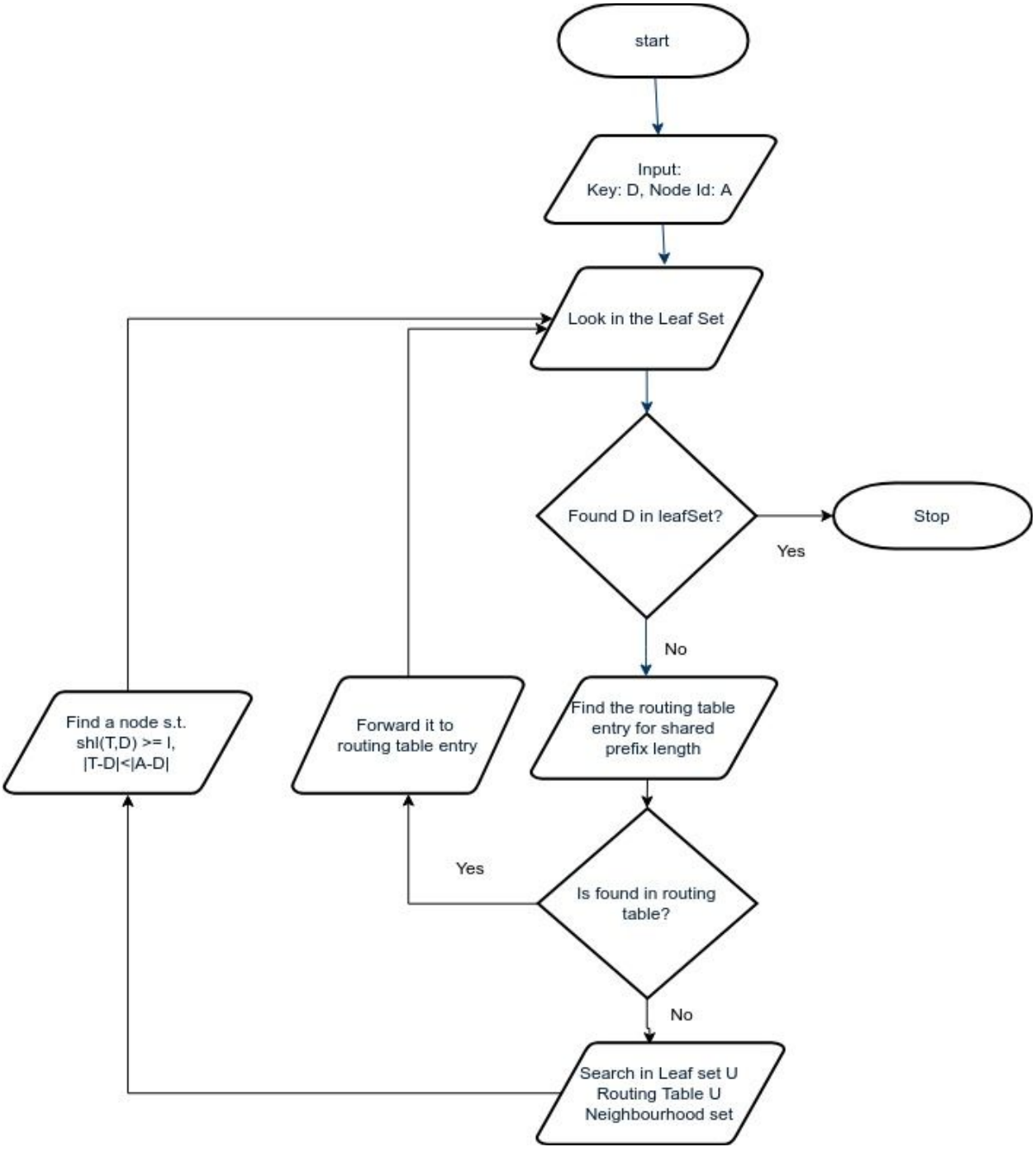

Fig. 3. Flow Chart representing Pastry Routing Algorithm

2.3. Design Summary

This section presents a brief summary for the overall design followed in our implementation. It discusses the driver program, important APIs that are essential to the Pastry core routing algorithm.

Driver Program:

The driver program reads the necessary arguments from the command line and initializes various parameters as part of the Pastry algorithm. The various parameters have been described in section 3.4. It initializes a bootstrap node which will provide assistance to any new nodes joining the Pastry network. It will provide the new node with a nearby node whom it can contact for sending the join request. In addition to the above, it will provide random nodes to clients querying for lookup as it maintains all the nodes information in the pastry network. This is done in DistAlgo as:

```
# Initialize the bootstrap node
boot = new(pastry.BootStrapNode)
setup(boot, (b, keySize, L, M, leafSetLen, tack))
start(boot)
```

The next step is to initialize the nodes in the Pastry network. The number of nodes is provided as an argument by the user. Various timing parameters are initialized during this phase. The code for the same in DistAlgo is:

```
# Initialize Pastry nodes
startTimeNodeInit = time.time()
ps = new(pastry.PastryNode, num=N)
totalTimeJoin = 0.0
```

Next, we initialize a controller to measure total user time, total system time, total process time, total memory and total wall clock time.
Using the below piece of code, we send wait for ‘JoinCompleted’ from all pastry nodes:

```
count = 0
for p in ps:
    endTimeNodeJoin = time.time()
    startTimeNodeJoin = time.time()
    setup(p, (ctl, boot, b, keySize, L, M, leafSetLen, ti, tack, tn))
    start(p)
    if await(some(received(('JoinCompleted'), from_= p))):
```

```
            count += 1
            endTimeNodeJoin = time.time()
            pass
        elif timeout(5):
            endTimeNodeJoin = time.time()
            pass
        time.sleep(0.1)
        totalTimeJoin += (endTimeNodeJoin - startTimeNodeJoin)

    # Only count nodes are initialized. Any node with same
    nodeid won't affect
    # count variable. Hence, N = count
    N = count
    endTimeNodeInit = time.time()
```

Now, N is the number of nodes which successfully joined the network.

Next, based on the user supplied parameter, we randomly kill dNodes and start lookup between any two nodes. We use the end keyword in DistAlgo to kill a process and get random nodes from bootstrap node.

For ClientLookup class, we get two random nodes from bootstrap node. We always make sure that bootstrap node returns us alive source and destination nodes based on inherent logic present in pastry.da line number 33 - 51. Then a request is sent to the source node and routing continues as per the algorithm presented in the paper. [1] The below code represents the snippet for initializing the ClientLookup class which performs lookup among two random nodes.

```
    # Start Lookup Process
    cl = new(ClientLookup, num=1)
    setup(cl, (boot, N, lTrials, tl))
    start(cl)
```

API Description:

Some of the important APIs that are implemented as part of Pastry core routing algorithm are -

1. *def route(message, dstObj, numHops, tt, tDist)* : Routes a message towards destination node ‘dstObj’. The number of hops to reach the current node is recorded in numHops. The distance traveled so far to reach the current node is recorded in tDist. tt is the lamport’s logical clock of the source node to distinguish lookups. This is used as there

can be conflict in the received variable provided in DistAlgo and same message may already be present in which case it won't initiate the lookup.

2. *def sendState(numHops, X):* This method is used to send the resulting state to a node X once a node joins the network. The number of hops is stored in numHops.

3. *def shl(A, B):* This is a helper function which calculates the shared prefix length between two nodes A and B.

4. *def makeListUnique(objSet):* This method acts as a helper function to filter out a list of objects and make them unique based on node Ids. It takes a list as an input and returns a list with unique node information in output. This is needed as we are passing a copy of the class PastryNodeInfo which contains basic information of each node. This copy cannot be filtered based on memory location as it may be different when it is passed to another node.

5. *update* and repair* functions:* These functions perform the act of updating / repairing the state tables of a node when they are corrupted or a new node information is provided to them.

The link to documentation generated using PyDoc is presented below:

https://github.com/unicomputing/pastry-distalgo/tree/master/PyDoc

The above link contains two files main.html and pastry.html which is the documentation generated for main.da and pastry.da respectively.

3. Implementation

3.1. Summary

The Pastry Algorithm as presented by Antony Rowstron and Peter Druschel [3] has been implemented using DistAlgo as the programming language. As DistAlgo is a very high level language for programming distributed algorithms and is extremely efficient it is chosen to implement the Pastry routing algorithm. In addition to the above, we could not find any open source implementation for Pastry in DistAlgo.

Each node in the Pastry network has a unique identifier (nodeId). When presented with a message and a key, a Pastry node efficiently routes the message to the node with a ndeId that is numerically closest to the key, among all currently live Pastry nodes. Pastry takes into account network locality; it seeks to minimize the distance messages travel, according to a scalar proximity metric like the number of IP routing hops. Moreover, Pastry is completely decentralized, scalable and self-organizing; it automatically adapts to the arrival, departure and failure of nodes.

Next, we will summarize the node arrival, departure / failures and routing algorithms that are used to make sure the network locality property is maintained.

Node Arrival: When a new node X arrives in the network, it needs to initialize its state tables. It initially contacts a node called BootStrap node to gain information of a node that is currently in the Pastry network which is nearby to X, according to the proximity metric. Let's call this nearby node A. Once node A is known, node X then asks node A to route a 'join' request across the network with key as X's nodeId. This routing takes place based on the routing algorithm described below. Once a node receives the join request, it sends its state tables to node X so that it can initialize itself and forwards the 'join' request to a node which may be nearer to X. In case, there is no such node present, the last node sends a 'done_join' request to the node X. Let's call this node Z. Once X receives all the states tables, it initializes its state tables and sends its resulting state tables to all the nodes in its leaf set, neighborhood set and routing table. This new node X initializes its leaf set based on the leaf set of node Z (as Z is closest based on the nodeId) and its neighborhood set based on the neighborhood set of node A (as A is the closest based on proximity metric as discussed before).

Routing: The pseudo-code of the routing algorithm is -

```
(1)  if (L_{-⌊|L|/2⌋} ≤ D ≤ L_{⌊|L|/2⌋}) {
(2)      // D is within range of our leaf set
(3)      forward to L_i, s.th. |D − L_i| is minimal;
(4)  } else {
(5)      // use the routing table
(6)      Let l = shl(D, A);
(7)      if (R_l^{D_l} ≠ null) {
(8)          forward to R_l^{D_l};
(9)      }
(10)     else {
(11)         // rare case
(12)         forward to T ∈ L ∪ R ∪ M, s.th.
(13)             shl(T, D) ≥ l,
(14)             |T − D| < |A − D|
(15)     }
(16) }
```

Fig. 4. Snippet of Pseudo Code as presented in [1]

A brief summary of the Pastry core routing algorithm is presented below -

Initially, a node searches for the key in its own leaf set. If the key is within the range of the leaf set, it forwards the request to the nearest node in the leaf set, otherwise it checks for an entry in its own routing table. The row is defined by the shared prefix length of the key and its nodeId and the column is defined by the i + 1th digit where i is shared prefix length. If the entry is none, then the node forwards the request to a the first entry in its leaf set + neighborhood set + routing table such that the following two conditions are satisfied

- Shared prefix length of the chosen entry's node id and key is greater than or equal to the shared prefix length obtained above and,
- T - D < A - D, where D is the key, A is the current node's nodeId and T is the node in consideration

Node Departure: Pastry nodes can arbitrarily fail at any point in time similar to any distributed nodes situated in a network. For handling node departures, we have assumed that the above assumption is true as it is a real world scenario. Once a node is no longer reachable, we attempt to repair the affected entries in leaf set and routing table on a lazy basis (i.e. when a failed node is contacted, it is repaired only at that time). The entries for neighborhood set are repaired both lazily and periodically where the time to repair the neighborhood set can be specified by the user. We assume that if we do not receive an acknowledgement from a node within a specific time, the entry can be deemed as failed and necessary steps are taken to repair the entries. The time can be provided by the user and is configurable. The repair process is carried out as follows -

- **Repairing Leaf Set:** When we try to contact an entry in the leaf set of a node and we do not receive an acknowledgement within a particular time, we first determine if the failed entry belongs to upper leaf set or the lower leaf set. Once we determine which entry has failed, we contact a live node with the largest index on the side of the failed node and request for its leaf set. Once we receive the leaf set, we can update our own leaf set otherwise the same process can be repeated. Unless all L / 2 entries fail simultaneously, (L being the leaf set length), we can repair the leaf set.

- **Repair Neighborhood Set:** When we try to contact an entry in the neighborhood set of a node and we do not receive an acknowledgement within a particular time, we request the neighborhood set from all the nodes in our neighborhood set and attempt to repair the entries by grouping only closest, live entries, based on proximity metrics.

- **Repair Routing Table:** When we try to contact an entry in the routing table of a node and we do not receive an acknowledgement within a particular time, a node first contacts first the node referred to by another entry Rli; i != d of the same row, and asks for that node's entry for Rld . In the event that none of the entries in row l have a pointer to a live node with the appropriate prefix, the node next contacts an entry Rl+1i; i != d, thereby casting a wider net. This procedure is highly likely to eventually find an appropriate node if one exists.

Thus, the above steps are followed during node arrival, routing and node departure in a Pastry network. The entire implementation is presented in the github repository below

https://github.com/unicomputing/pastry-distalgo

3.2. Development Statistics

Programming Language: DistAlgo (for Pastry network), Python (for generating graphs)
Code Size: 1015 (pastry.da) 200 (main.da), 194 (generateGraph.py)
Development Effort: 30+ hours, 4 weeks
Documentation Tool: PyDoc
File Size: 49.2 KB (pastry.da), 7.04 KB (main.da), 5.29 KB (generateGraph.py)
Operating System: Ubuntu 18.04.1 LTS
DistAlgo Version: 1.0.12
Python Version: Python 3.6.7
Processor: Quad-core x86_x64 processor with Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz

3.3. Comparison with previous implementations

3.3.1. Comparison with Python implementation by Kapil Thakkar

During Assignment 5, we found an implementation by Kapil Thakkar presented in python that follows the routing as described by Antony and Rowstron and Peter Druschel [1]. Although the documentation is not appropriate, we found the code clarity as one of the major factors for regarding this implementation as the best implementation. The functions and variables used within the python code clearly represent the steps as presented in the routing algorithm. However, we found the following diversions from the actual implementation presented as pseudo-code in the paper.

1. The entire algorithm is sequential. However, in reality the Pastry network is completely decentralized.

2. The functions "isEligibleDownLeaf" and "isEligibleUpLeaf" are improper. A node is considered to be placed in up leaf or down leaf based on node's nodeId itself. There is no correlation with shared prefix length. However, Kapil has mentioned a correlation between them in helper.py line number 136 and 141 which we believe is wrong. Similar explanation holds true for

3. In the function "def updateLeafSet(Z, X):" in operations.py line number 100, Kapil adds the leaf set of node Z directly to leaf set based on its eligibility which is not in accordance with the paper. In reality, the leaf set should contain the closest nodes based on the node Ids unlike Kapil's implementation in which the nodes are directly added without checking. There can be a case in his implementation where some nodes in leaf set should belong to upLeafSet or downLeafSet as they are closer than existing nodes but are ignored as the leaf set is currently full.

4. In the function "def updateNeighborSet(A, X):" in operations.py line number 91, the neighborhood nodes should be appended based on the relative distance among the nodes in the current node's neighborhood set and that of A's neighborhood set. Kapil has correctly implemented this logic during repairing neighborhood set line 347. The same logic should be applied here in order for the algorithm to closely follow the paper.

5. In the function "def updateRoutingTable(A, X, routePath):" in operations.py line number 78, if the entry is already present in the routing table, then the entry that should belong to the routing table should be closer to the current node, based on scalar proximity metric. This is mentioned in page 4, second paragraph first line in the paper.

3.3.2. Comparison with DistAlgo implementation by Ken

As part of our project, Professor Annie Liu had given us a pastry implementation (in DistAlgo) by Ken [3]. After carefully reviewing the implementation and comparing with several aspects of the Paper presented by [1], we found the following diversions from the actual implementation presented as pseudo-code in the paper.

1. The proximity function used by Ken is not Euclidean i.e. Triangulation property doesn't hold among Pastry nodes. It is mentioned in the paper that basic routing may not be affected, however, locality properties may suffer. Hence, we have used a proximity function that is Euclidean which more closely aligns with how the nodes are treated in the paper.

2. When a node failure happens in Pastry network , Ken currently broadcast this information to all the nodes in his leaf set, neighborhood set and routing table. Although this is one way of handling departures, however, nodes in a Pastry network may fail or depart without warning. We are trying to address this problem currently so that it closely follows the pape.

3. In Ken's implementation, the part handled by line number 479 - 482 (node2.da) imparts that it looks for a node with highest Id which is less than key (NodeId), but this is not in accordance with the pseudo code presented in the paper. In paper it is clearly mentioned that we look for that node which has least absolute distance (|D-Li| is minimal as per line 3 of pseudo code of routing algorithm paper).

   Following is the example which supports our claim:

   Suppose a node's (Having nodeId : 2568) upper leaf set contains two nodes with node Ids 2578 and 2598. Now, if we want to route D = 2597. Then according to Ken's implementation D = 2597 will be forwarded to node 2578 which is incorrect as it should to be forwarded to node 2598 as it has least absolute distance (|2597 - 2598| < | 2597 - 2578|. Hence 2597 is better option than 2578 for D = 2597).

   Similar counter-example can be given for lower leaf set code presented in 497 - 499 (node2.da) of Ken's implementation.

4. In Ken's implementation, the part handled by line number 506 - 530 (node2.da) represents lookup for entry in routing table which is not in accordance with line 6-8 (lookup for entry in routing table) of pseudo code of paper.

(5) // use the routing table
(6) Let $l = shl(D, A)$;
(7) if ($R_l^{D_l} \neq null$) {
(8) forward to $R_l^{D_l}$;

Fig. 5. Snippet of Pseudo Code for look up in routing table [1]

It is easy to understand that, we just need to check one entry in the routing table and forward the message to that entry if it's not null. Instead of checking only one entry in routing table, Ken is traversing entire row of routing table(line 518 in node2.da). We believe that Ken has to traverse the entire row because of maintaining(updating) routing table in inappropriate way (different than approach mentioned in the paper). We have discussed this issue in next point.

5. In Ken's implementation, the function "def addToRoutingTable(node, nodeid):" which is responsible for adding a node in the routing table handled by line 611 - 627 (node2.da) is not in accordance with the paper. It violates the property by which we maintain the routing table entries. For example, for b = 2, keySize = 16, a node with nodeId 10233102 might have the following entries in its routing Table.

NodeId 10233102

Leaf set	SMALLER	LARGER	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

Routing table

-0-2212102	**1**	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	**2**	10-3-23302
102-0-0230	102-1-1302	102-2-2302	**3**
1023-0-322	1023-1-000	1023-2-121	**3**
10233-0-01	**1**	10233-2-32	
0		102331-2-0	
		2	

Fig. 6. Snippet of Pastry Node as mentioned in [1]

Suppose, this node wants to add a new node with nodeId 12331023 in its routing table. According to Ken's implementation "addToRoutingTable" function, table's 2nd entry will be:

0	1-1-301233	1-2-230203	1-2-331023

Here last column will have a wrong entry. For this row, last column entry's 2nd digit must be '3'. Hence, this entry violates basic routing table property.

6. In routing algorithm, when node fails to find appropriate entry from leaf set or routing table, it follows the following steps:

(11) // rare case
(12) forward to $T \in L \cup R \cup M$, s.th.
(13) $shl(T, D) \geq l$,
(14) $|T - D| < |A - D|$

Fig. 7. Snippet of Pseudo code from [1] for rare case routing

The part handled by lines 561 - 602 (node2.da) in Ken's implementation, are representation of lines 11-14 of pseudo code of routing algorithm. When we looked through the implementation lines (582-602), we found that it does not follow the pseudo code closely. In the implementation, after sorting all possible nodes (L U R U M) by prefix length with current node's nodeid, the last node in list will be selected and two conditions (steps 13-14 in above figure) will be tested against it. If it passes these conditions, message will be forwarded to it. Otherwise, lines 582-602(node2.da) will be executed to forward the message. This approach is not according to the routing algorithm mentioned in the paper. Instead of checking only last node in list, we must check all other nodes also that might satisfy these two conditions and forward the request to first node that satisfies it. If there are no such nodes that satisfy the above conditions, then only current node should handle the message itself. Furthermore, instead of checking for condition |T - D| < |A - D| it checks for |T - D| > | A - D| which we believe is a logical error from Ken.

7. Ken wrote code in lines 540 - 556 to avoid "infinite loop". We found out that there's no need of these lines if implementation follows paper correctly (we verified it in our implementation) because, in many runs including different parameters, we didn't face this "infinite loop" issue. There's no mention of this issue even in the paper [1].

3.4. Usage

The program takes in 11 arguments in the same order. The default values are included in square brackets in the end.

N: Number of Pastry nodes in the network [Initial value: 10]
dNodes: Maximum number of nodes that should be failed eventually [Initial value: 0]
lTrials: Number of trials for performing lookup [Initial value: 50]
b: Configurable Parameter, usually default to 4 [Initial value: 2]

keySize: Size of nodeid in bits [Initial value: 16]
tl: Timeout after which lookup is deemed failed [Initial value: 15]
td: Timeout after which deletion of Pastry node is deemed failed [Initial value: 1]
ti: Timeout after which initialization of a Pastry node is deemed failed [Initial value: 1]
tack: Timeout to wait for acknowledgement of a message [Initial value: 1.0]
tn: Time to periodically check neighborhood set [Initial value: 5]
tps: Timeout for determining the liveness property [Initial value: 100]

Once the dependent programs are compiled using

python -m da.compiler fileName.da

We can simply run

python -m da --message-buffer-size mbSize main.da

The dependent programs are pastry.da and controller.da. Pastry.da contains the entire pastry implementation, controller.da is used to measure running times for the program and main.da is the driver program. We have used mbSize as 32000 to tackle the MessageTooBigException during routing of messages.

We have created a shell script which can be run directly from the shell using

./make.sh

The controller is disabled by default as it takes a lot of time for initialization of nodes due to message passing overhead to the controller. It is used to measure running times and other metrics of the implementation. When required, the controller can be enabled by uncommenting controller.run in line number 186 in pastry.da and lines 132 - 134 in main.da which initializes the controller.

4. Testing and Evaluation

4.1. Graphs

4.1.1. Average number of hops vs Number of nodes

This experiment was done to measure performance of the implementation. As mentioned in the paper [1], the average routing hops will be less than $\log_{2^b}(N)$. Here, x-axis represents number of nodes and y-axis represents average number of hops required for particular network size (number of nodes). The results also shows that number of hops increases with network size as expected.

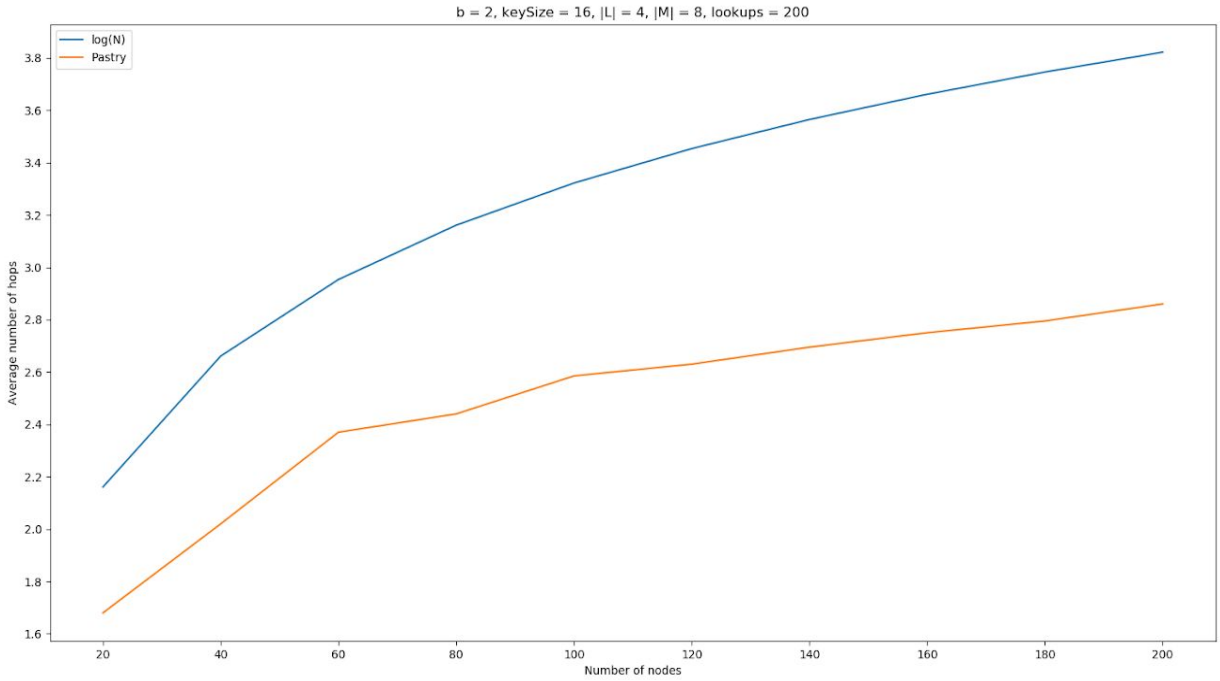

Fig. 8. Average number of routing hops versus number of Pastry nodes, b = 2, keySize = 16, |L| = 4, |M| = 8 and 200 lookups.

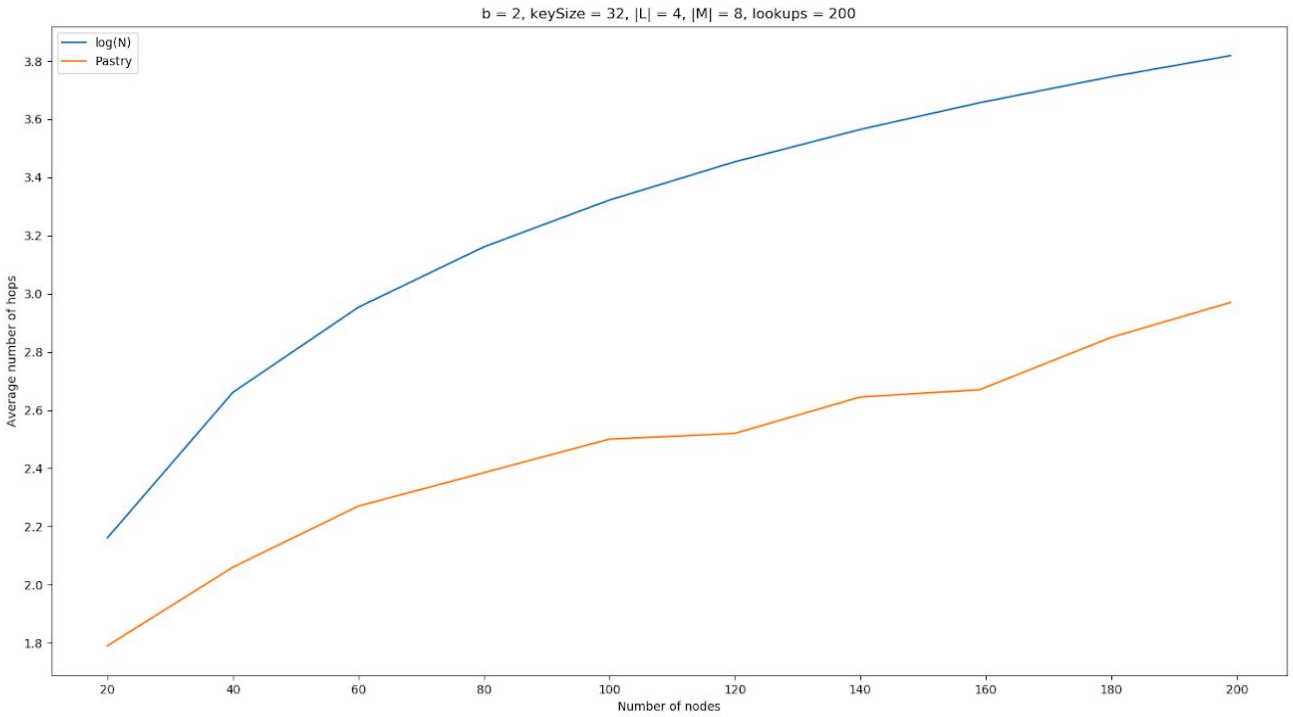

Fig. 9. Average number of routing hops versus number of Pastry nodes, b = 2, keySize = 32, |L| = 4, |M| = 8 and 200 lookups.

4.1.2. Probability vs Number of routing hops

Figure 10 and 11 represents the distribution of number of routing hops taken for N = 200. We can see that with very high probability required number of hops will be less than $\log_{2^b}(N)$ (4 in this case). In page 6, fourth paragraph of the paper [1], the authors have mentioned that " Analysis shows that with |L| = 2b and |L| = 2 × 2b, the probability that this case arises during a given message transmission is less than .02 and 0.006, respectively. When it happens, no more than one additional routing step results with high probability". The figures 10 and 11 show that, at any point in time, the expected number of hops do not exceed 5 which is according to the results presented in the paper.

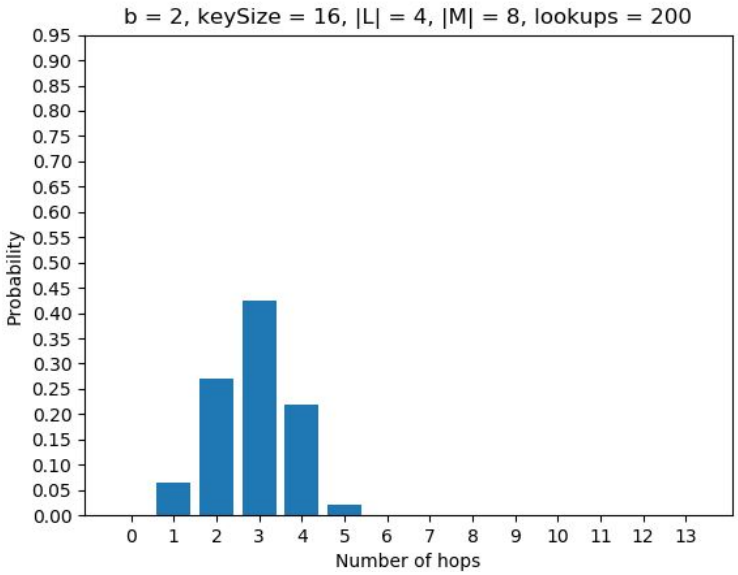

Fig. 10. Probability versus number of routing hops, b = 2, keySize = 16, |L| = 4, |M| = 8, N = 200 and 200 lookups

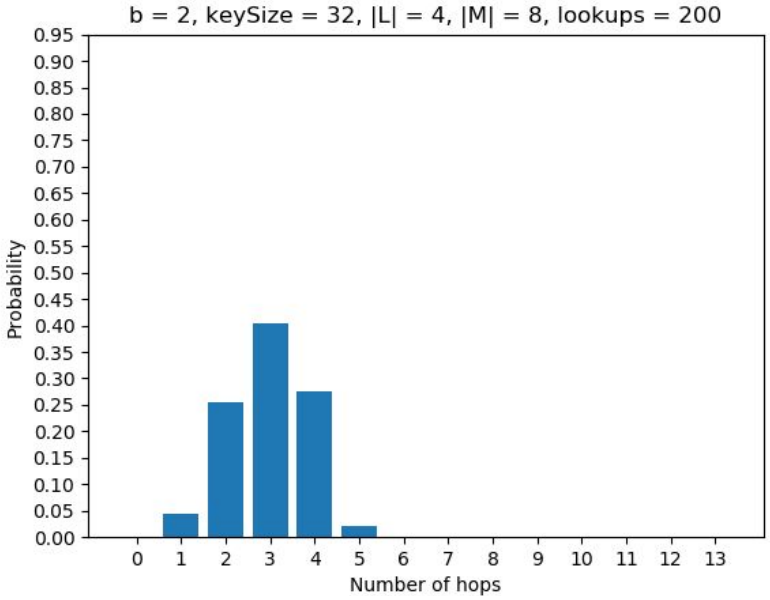

Fig. 11. Probability versus number of routing hops, b = 2, keySize =32, |L| = 4, |M| = 8, N = 200 and 200 lookups

4.1.3. Route Distance vs Number of Pastry nodes

This experiment is done to evaluate locality properties of pastry algorithm. We are comparing the distance traveled by a message (according to proximity metric) using Pastry with the distance the message would have traveled if a node had complete routing tables (fictitious routing scheme). The goal is to compare the trade off between size of routing tables maintained by Pastry nodes and distance traveled by a message. Figure 12 and 13 shows that distance traveled by a message is 70%- 80% higher than fictitious scheme (“complete routing table”).

Considering routing table entries (approximately $\log_{2^b}(N) * (2^b - 1) = 12$, N = 200) in Pastry with fictitious scheme with 199 entries(for N=200) in routing table, results are quite good.

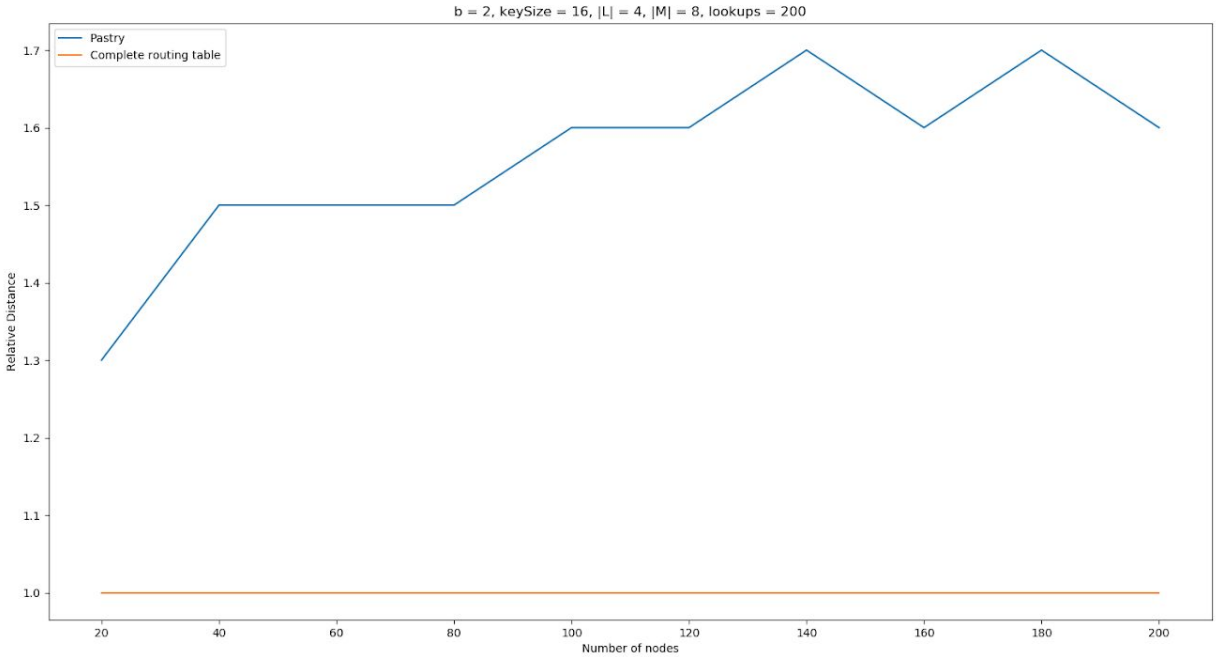

Fig. 12. Route distance versus number of Pastry nodes, b = 2, keySize = 16, |L| = 4, |M| = 8, and 200 lookups.

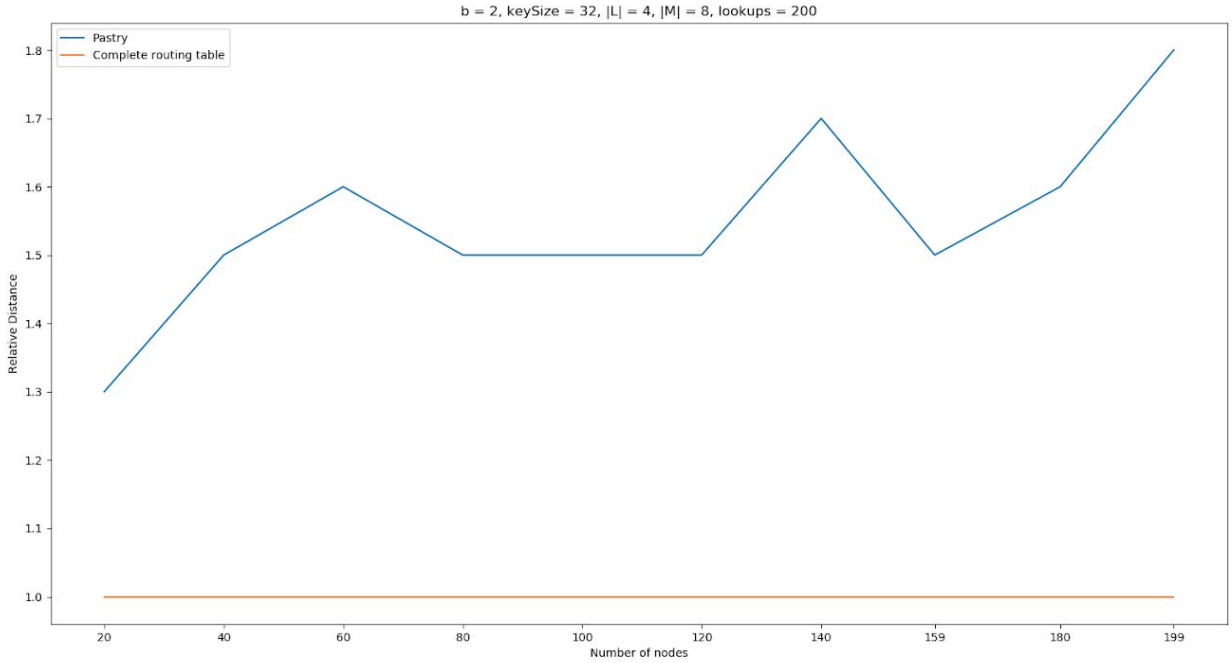

Fig. 13. Route distance versus number of Pastry nodes, b = 2, keySize = 32, |L| = 4, |M| = 8, and 200 lookups.

4.1.4. Number of routing hops vs Node failures

This experiment is done to see the effect of repairing routing tables (in case of node failures) on number of routing hops. We failed 10 (5%), 20 (10%), 40 (20%) nodes out of total N = 200 nodes. In all these three scenarios, we measured average number of hops for three cases 1) No failure 2) Failure with no routing table repair 3) Failure with routing table repair. Following figures show that if repairing routing table functionality is disabled, the performance of pastry is bad compared to normal case (no node failures), average number of hops is high. But, if repairing routing table functionality is up and running, the performance of Pastry is almost same compared to normal case (no node failures). We could also see that as number of node failures increases, performance of Pastry degrades in case of no repairing of routing table.

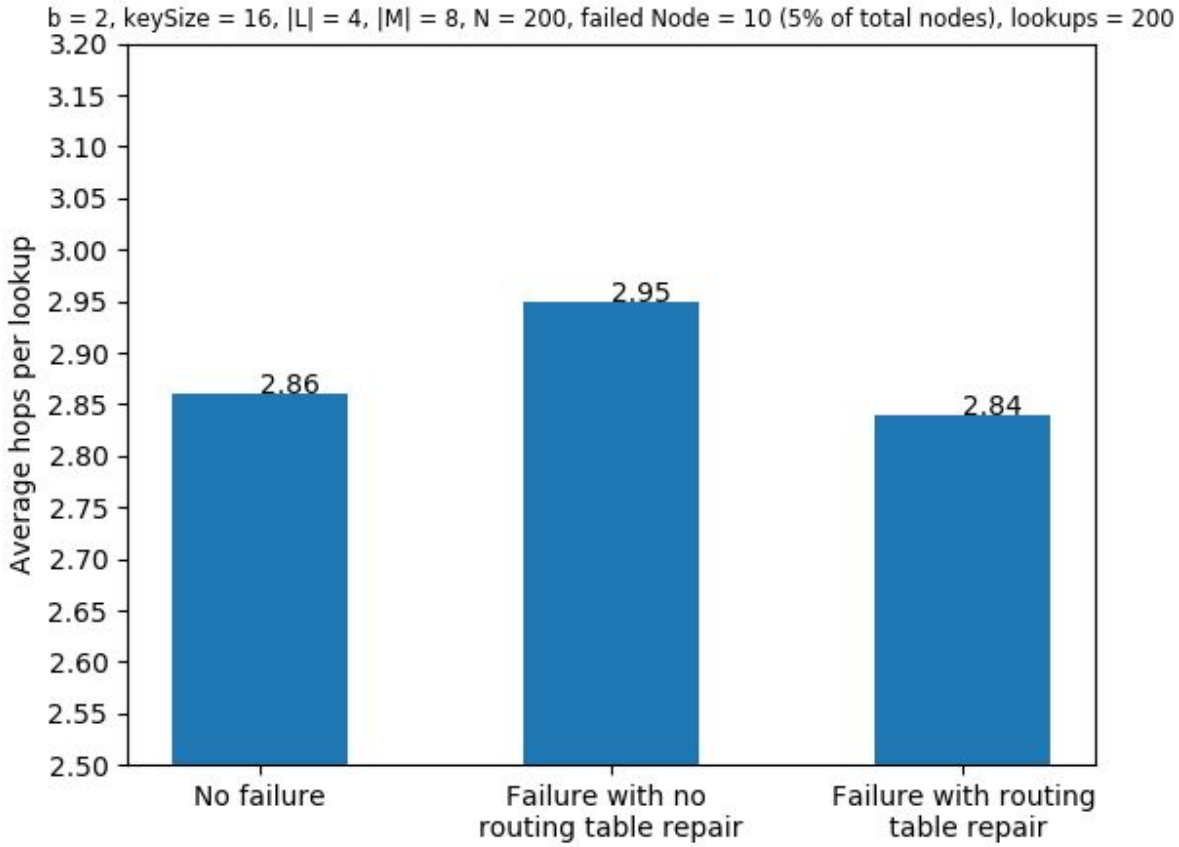

Fig. 14. Average number of routing hops versus node failures, b = 2, |L| = 4, |M| = 8, 200 lookups and 200 nodes with 10 failing.

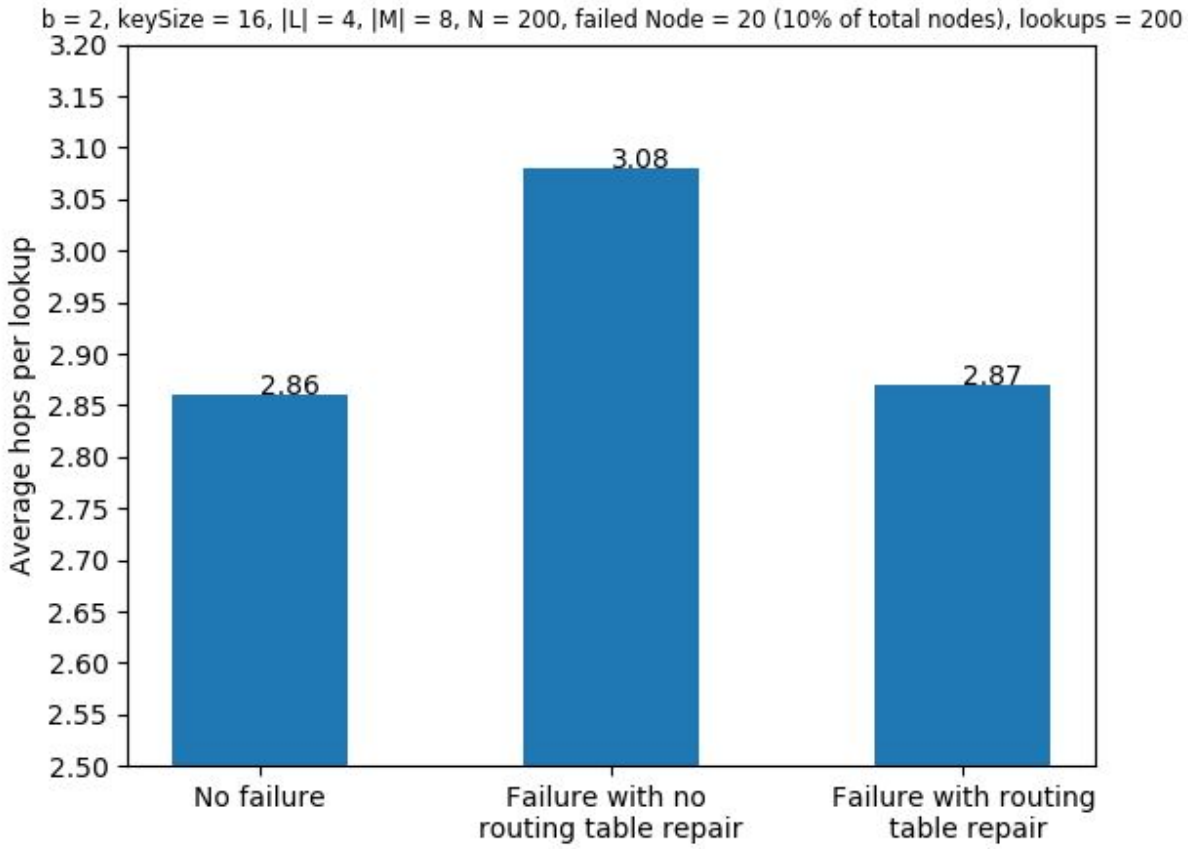

Fig 15 Average number of routing hops versus node failures, b = 2, |L| = 4, |M| = 8, 200 lookups and 200 nodes with 20 failing.

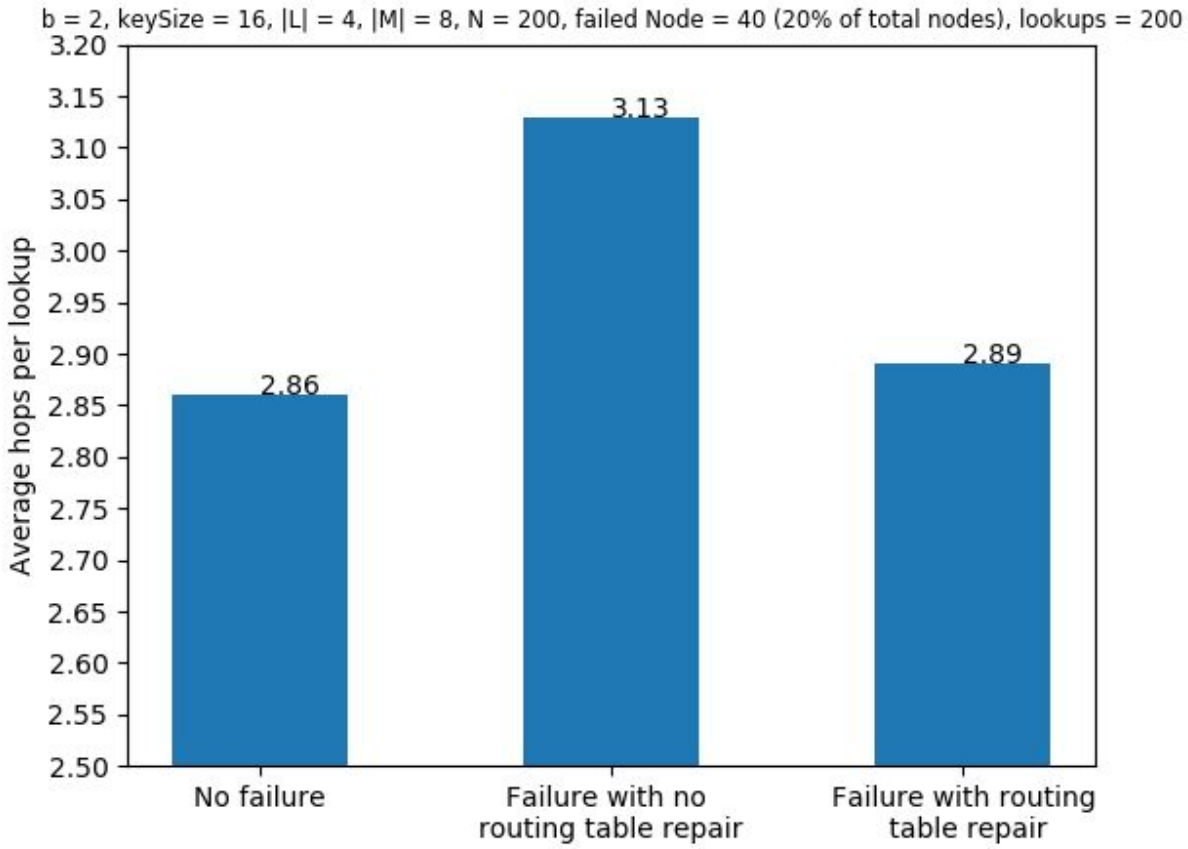

Fig. 16. Number of routing hops versus node failures, b = 2, |L| = 4, |M| = 8, 200 lookups and 200 nodes with 40 failing.

4.1.5. Total node join time vs Number of Pastry nodes

This experiment was done to measure performance of the implementation. The node join time is the time when 'join' request is sent to the closest neighbor derived from BootStrap node to the time when the node X is able to route and receive messages in the Pastry network. In page 8, second paragraph of the paper [1], it is mentioned that once the node X sends its resulting state to other nodes in its state tables, it is ready to participate in routing process and receive messages.

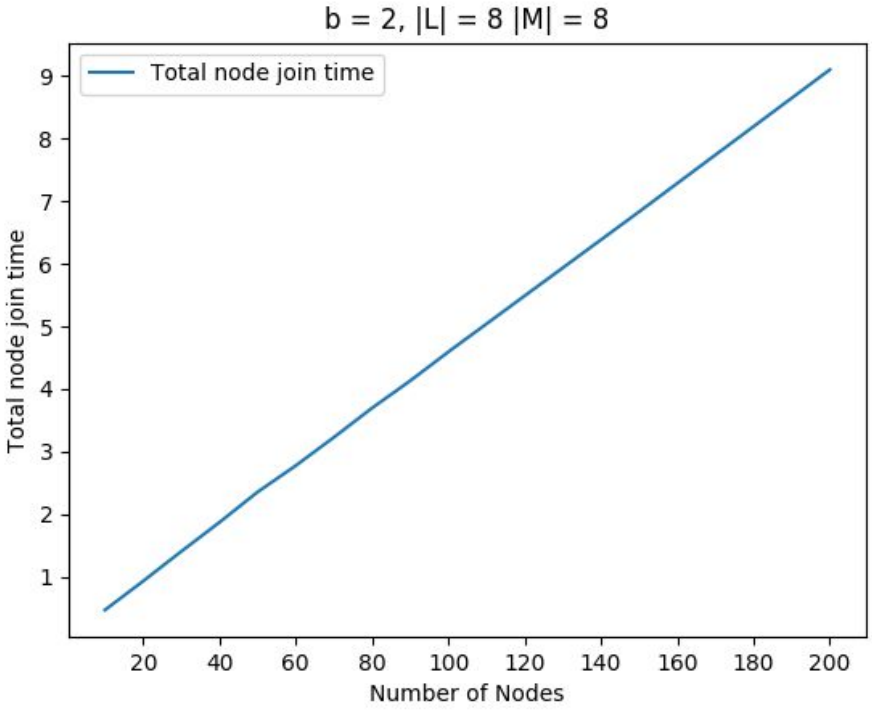

Fig.17. Total Node Join Time (sec) vs Number of Nodes for b = 2, |L| = 8 and |M| = 8

4.1.6. Average node join time vs Number of Pastry nodes

This is the average node join time and is a measure of performance of the implementation.

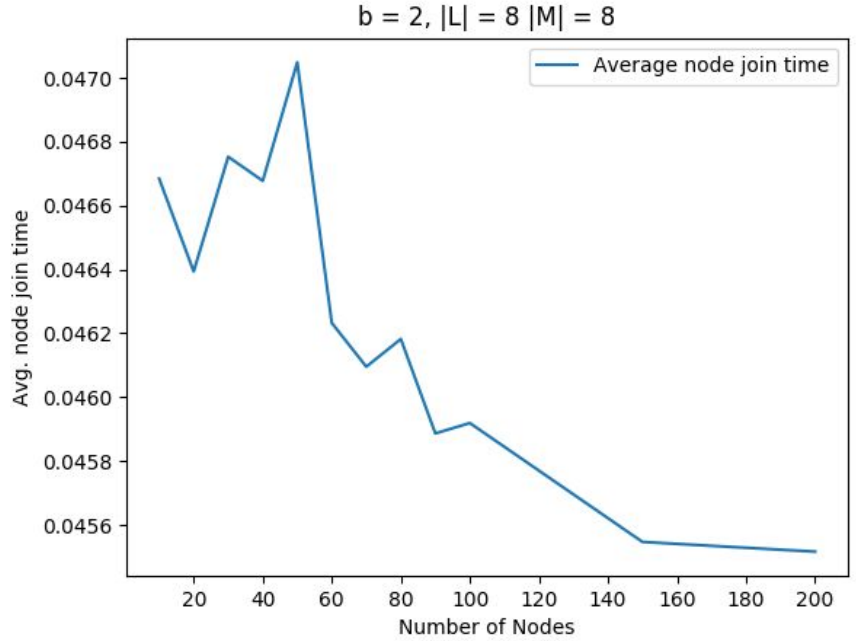

Fig.18. Average Node Joining Time (sec) vs Number of Nodes for b = 2, |L| = 8 and |M| = 8

4.1.7. Node initialization time vs Number of Pastry nodes

Node initialization time is time when the 'JoinCompleted' message is received by the node X. It is a measure of the performance of the implementation. It defines the total time taken after which new node can join the Pastry network and start the initialization process. We can see that there is a linear increase in the running times as the number of nodes in the network is increased.

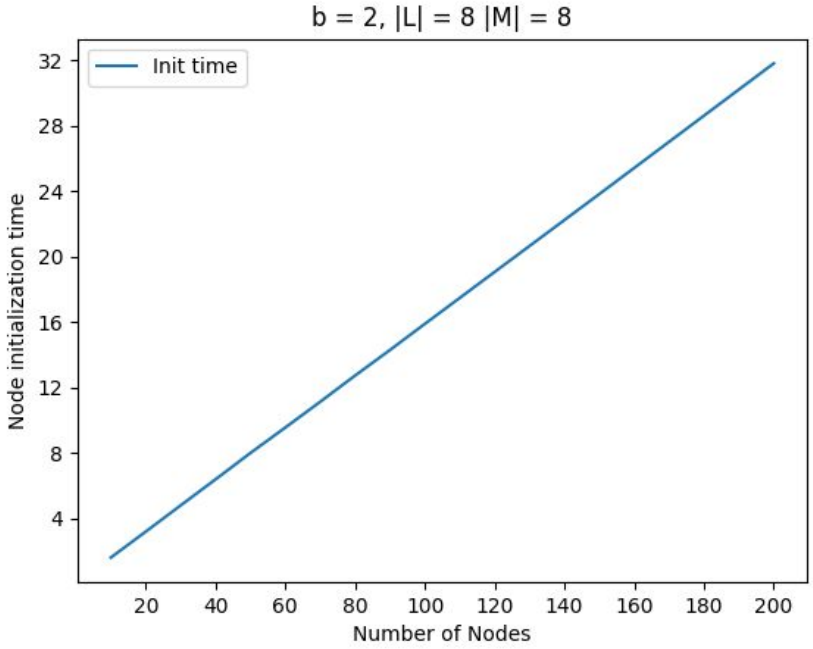

Fig 19. Node Initialization Time(sec) vs Number of Nodes for b = 2, |L| = 8 and |M| = 8

4.1.8. Running times vs Number of Pastry nodes

The controller is used to measure the running times of the current implementation. It is a performance measure of the current implementation. This graph presents total user time and total process time vs number of nodes in the pastry network. We can see that there is a linear increase in the running times as the number of nodes in the network is increased.

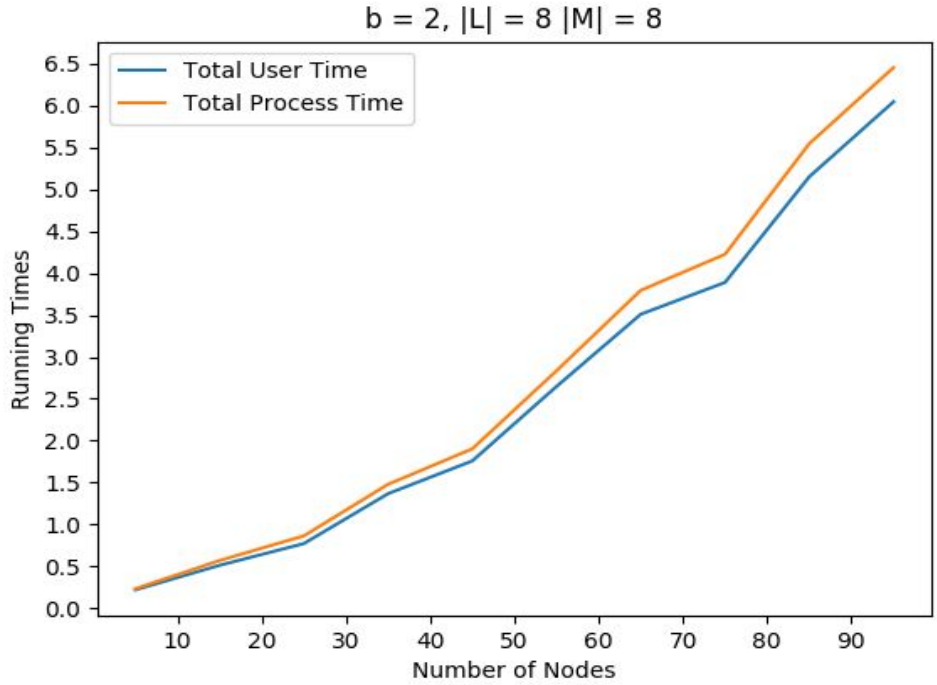

Fig. 20. Running Time(sec) vs Number of Nodes for b = 2, |L| = 8 and |M| = 8

4.1.9. Total system time vs Number of Pastry nodes

The controller is used to measure the total system time of the current implementation. It is a performance measure of the current implementation. This graph presents total system time vs number of nodes in the pastry network. We can see that there is a linear increase in the running times as the number of nodes in the network is increased.

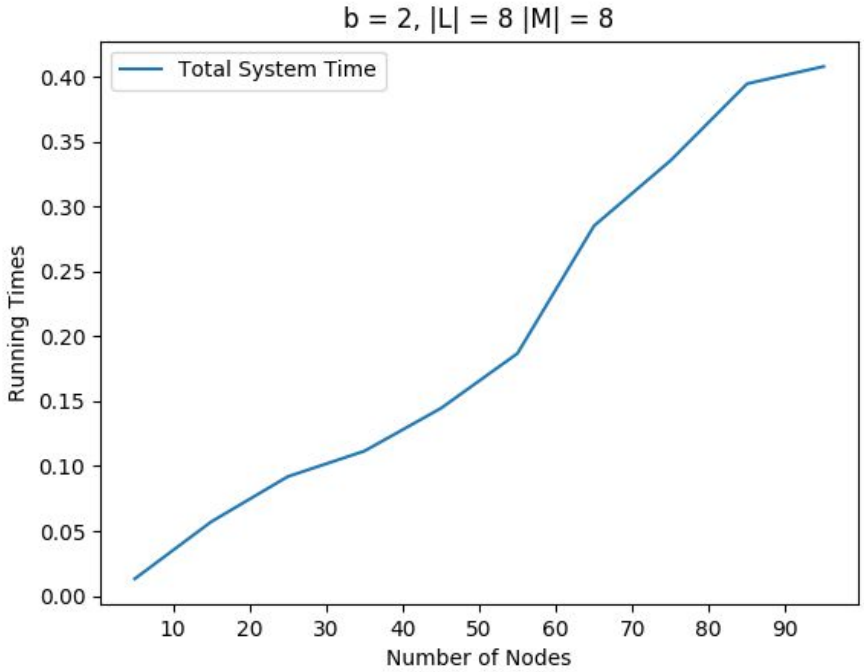

Fig. 21. Total System Time(sec) vs Number of Nodes for b = 2, |L| = 8 and |M| = 8

4.1.10. Total memory vs Number of Pastry nodes

The controller is used to measure the total memory for the current implementation. It is a performance measure of the current implementation. This graph presents total system time vs number of nodes in the pastry network. We can see that there is a linear increase in the running times as the number of nodes in the network is increased.

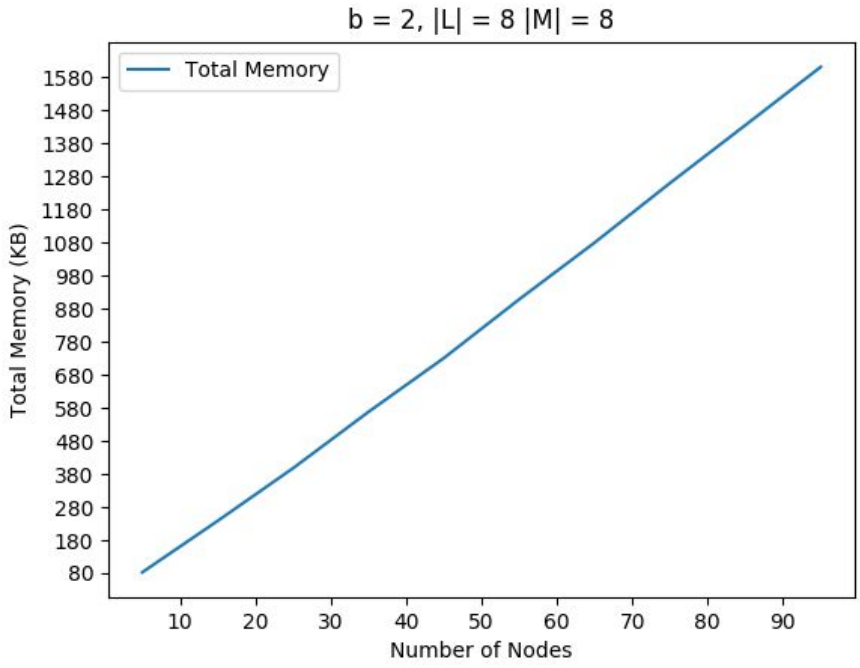

Fig. 22. Total Memory (KB) vs Number of Nodes for b = 2, |L| = 8 and |M| = 8

4.1.11. Total wall clock time vs Number of Pastry nodes

The controller is used to measure the wall clock time for the current implementation. It is a performance measure of the current implementation. This graph presents total system time vs number of nodes in the pastry network.

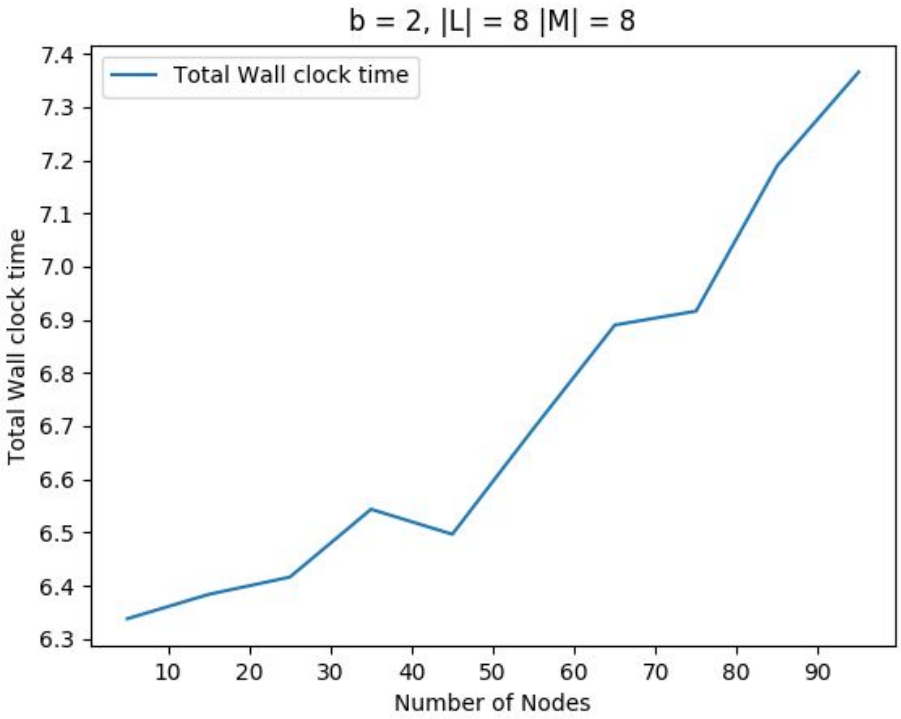

Fig. 23. Wall Clock time (sec) vs Number of Nodes for b = 2, |L| = 8 and |M| = 8

4.2. Comparison with existing implementations

Section 1.2 discusses several implementations of Pastry algorithm. We selected a best implementation from the above based on several factors presented in section 1.2 and 3.3. After studying various implementations for Pastry algorithm presented in paper [1], we come to a conclusion that the implementations had significant diversions from the actual paper. On comparison with two best implementations, we enumerated the deviations / digressions in section 3.3 for both implementations. For our implementation, we calculated various performance metrics like total user time, total system time, total process time, total memory and total wall clock time in addition to other metrics presented in the paper [1]. We haven't compared the running times, memory, CPU times e.t.c. with other implementations as they are not closely following the paper. We believe that this implementation will serve as a foundation for measuring and comparing all the above mentioned metrics. Any further implementation of Pastry core routing algorithm in different programming languages can refer to our implementation and its results to compare the running times, memory, CPU times e.t.c.

4.3. Summary

To summarize the above results, we see that the average number of hops without failure, failure with no repair and failure with repair are as described in the paper [1]. The results are shown in the section 4.1.4. If no failures occur, the average number of hops for with N = 200, b = 2,

keySize = 16 is 2.86 which is less than $\log_4$ 200 as mentioned in page 6, third paragraph of the paper [1]. As we can see, the average number of hops is almost same as that before the failures. This is in accordance with Figure 10 of the paper [1]. In case of no repair, the code mostly ends up in lines 11 - 14 [1] of the pseudo-code presented in Table 1 which is a rare case if all the routing tables are perfectly formed and no failures occur. Hence, the average number of hops in case of no repair jumps to 2.95.
The paper [1] states that expected number of routing steps is $\log_{2^b}$ N assuming accurate routing tables and no recent node failures (page 6, paragraph 3). This result is also being exhibited in the graph presented in section 4.1.1. The above results show that our implementation closely follows that of the prototype implementation presented in the paper [1] as the results are similar. We measure only some of the metrics presented in the paper due to time constraints and other factors, the details of which are presented in the section 4.2.

The paper has considered values of b = 4, N ranging between 1000 to 100,000, |L| = 16, |M| = 32, keySize = 128 and lookups = 200,000. However, due to machine limitations we have considered value of N to be less than 200 and lookups to be less than 200. The paper has only considered these values for performing experiments, hence we have not varied these variables to keep the results consistent with the paper. We haven't varied timeouts for lookup (t_l), delete (t_d), init (t_i) and acknowledgement (t_{ack}) as ideally any of the nodes shouldn't timeout during any of the above phases. In addition, varying timeout to the acknowledgement (t_{ack}) will only vary the waiting time for a node to receive the acknowledgement if any node is unreachable. Hence, varying this parameter is not useful in any of the measurements and context of the paper. Moreover varying lookup trials is also not useful as it is intended to take average for all the various lookups performed. Varying t_n, which is the time for periodically checking inconsistencies in neighborhood set is also futile as we are performing repair of nodes in neighborhood set lazily as well. We periodically check to reduce the overhead when such a node is discovered lazily. We also do not vary the number of lookup trails (lTrials) as it used for taking an average for calculating average number of routing hops with / without failures.

5. Conclusion

We conclude that some of the experimental results presented in the paper by Antony Rowstron and Peter Druschel [1] hold true when the implementation was carried out at a small scale in DistAlgo. Due to machine limitation, the scope for increasing number of nodes and performing lookups was limited. The section 3 in the paper [1] illustrates that the experiments were performed on a quad-processor Compaq AlphaServer ES40 (500MHz 21264 Alpha CPUs) with 6 GBytes of main memory, running True64 UNIX, version 4.0F. The Pastry node software was implemented in Java and executed using Compaq's Java 2 SDK, version 1.2.2-6 and the Compaq FastVM, version 1.2.2-4. We have run the experiments on quad-core x86_x64 processor with Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz, running Ubuntu 18.04.01 LTS. Hence, the prototype implementation presented in the paper is testing the implementation with 100000 nodes and 200000 lookups. However, we have limited the number of nodes and lookups to 200 each. At a small scale, we were able to exhibit the same results as that of the prototype implementation presented in the paper. We currently only carried out correctness and performance testing for routing performance (section 3.1) and node failures (section 3.4) as presented in paper [1].

We determined that for values like b = 4, keySize = 128 and N = 40, the average number of routing hops is greater than $\log_{2^b} N$. Ideally, it is presented in the paper that [1] the average number of routing hops is less than $\log_{2^b} N$. We believe the reason behind this is the unequal distribution digits in the nodeId. In page 4, second paragraph of the paper, it is mentioned that "The uniform distribution of nodeIds ensures an even population of the nodeId space; thus, on average, only log2bN rows are populated in the routing table". In the above scenario, as the nodeIds are randomly generated and number of nodes are less, we cannot achieve this uniform distribution which is a plausible reason for this digression.

The results validation for maintaining the network (section 3.2) and replica routing (section 3.3) are left as future work due to time constraints and other factors which is discussed in brief in the next section.

6. Future Work

Due to time constraints and other factors discussed below, we were only able to validate the results for section 3.1 and 3.4 of the paper [1].

- The section 3.2 (Maintaining the network) and section 3.3 requires us to develop three different variants of the same algorithm (namely SL, WT and WTF) and compare them and obtain the graphs presented in Fig. 7 and 8 in the paper [1]. As we did not had time to develop all the variants, hence, this is left for future work.
  In addition to the above, we were not able to plot the graph for Fig. 9 (Quality of routing tables) in the paper [1]. For plotting this graph, a specific distinction needs to be made between Optimal and Sub-Optimal entries. Due to stringent time constraints and amount of code change needed to incorporate the above behaviour, we are leaving this for future work.

- Currently, we assume that the source and destination nodes are always alive. There might be cases that destination node is currently not alive in which case proper notification should be sent to source node informing the same.

7. References

1. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems, Antony Rowstron and Peter Druschel, 2001 https://link.springer.com/chapter/10.1007/3-540-45518-3_18#citeas
2. Pastry implementation by Kapil Thakkar in python https://github.com/kapilthakkar72/Pastry-and-Chord-DHT/tree/master/Implementation/Pastry
3. Pastry implementation by Ken Koch in DistAlgo https://github.com/unicomputing/pastry-distalgo-2012-Ken-Koch
4. DistAlgo documentation https://sites.google.com/site/distalgo/
5. Pastry implementation in Java by Caleb https://github.com/ctebbe/P2P-Pastry-Implementation
6. Pastry implementation in Java by Hamersaw https://github.com/hamersaw/BasicPastry/tree/master/src/main/java/com/hamersaw/basi%20c_pastry
7. Pastry implementation in C++ by ngrj93 https://github.com/ngrj93/Pastry
8. PAST: a large-scale, persistent peer-to-peer storage utility, P. Druschel and A. Rowstron, May 2001 https://ieeexplore.ieee.org/document/990064
9. SCRIBE: A large-scale and decentralised application-level multicast infrastructure, M. Castro, P. Druschel, A.-M. Kermarrec and A.I.T. Rowstron, December 2002 https://ieeexplore.ieee.org/document/1038579
10. POST: A Secure, Resilient, Cooperative Messaging System,Alan Mislove, Ansley Post Charles Reis, Paul Willmann and Peter Druschel http://www.ccs.neu.edu/home/amislove/publications/POST-HotOS.pdf
11. Pastry implementation in Python by MuxZeroNet https://github.com/MuxZeroNet/pastry
12. Pastry implementation in C++ by Danvil https://github.com/Danvil/pastry
13. CSE 535 course webpage and lecture slides http://www3.cs.stonybrook.edu/~liu/cse535/
14. Our Pastry implementation https://github.com/unicomputing/pastry-distalgo