

OS PROJECT

PASTRY Distributed Hash Table

Team Name : 4 Nodes

Juhi Gupta	(20172031)
Kaushik LV	(20172138)
Rishabh Murarka	(20172111)
Krishna Dwypayan	(20172076)

Introduction

In P2P systems, networks organise themselves into overlay networks and relay/store data for each other. An overlay network is a network that is built on top of another network. Nodes in an overlay network can be thought of as being connected to each other by virtual or logical links, each of which is corresponding to a physical path in the underlying network. In distributed systems such as P2P networks and client-server applications, the nodes run on top of Internet.

Pastry is a structured type of an overlay network, structured in the sense that it imposes particular structures on the overlay networks. The structuring is necessary to achieve efficient resource search in such large scale distributed storage networks. The structure imposed by Pastry is that of Distributed Hash Tables. Like any other structured P2P overlay, Pastry maintains a structure to accommodate the participating nodes and their corresponding routing data, aside to the routing algorithm to be followed by the nodes to enable communication. Also maintained is a structured format of join/leave mechanism that ensures fault tolerance and enables self-organisation.

Distributed Hash Tables apply a hashing function (like the Secure Hash Algorithm version 1, SHA-1), to form the node IDs. IDs for the data items are created by applying the same hash function. The node IDs and data IDs fall into the same address space. These data items are stored on the closest node with the node ID greater than or equal to the data ID. If the node with the closest node ID does not store the data item, then the node does not exist in the system. Following the respective data storage tables, any node in the system can be found in the overlay.

P2P systems are highly dynamic, with respect to the nodes that can join or leave at anytime during the functioning of the overlay. For a node to join, it needs to acquire a unique node ID and then position itself into the overlay structure based on the node ID and the geometry of the

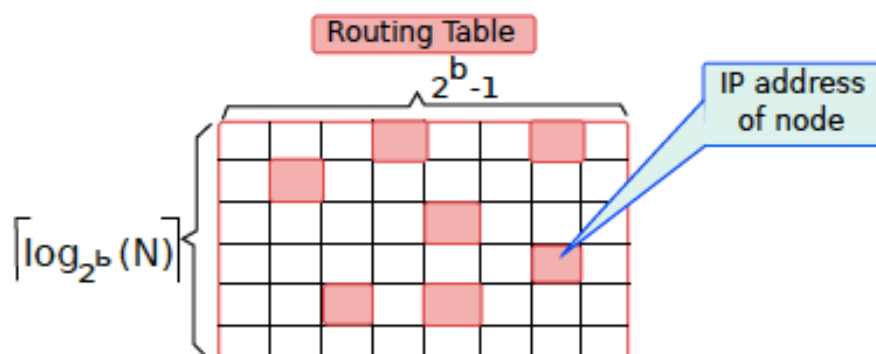
system. Subsequently, all the routing tables of the nodes affected by the join of the new node and the routing table of the joining node itself are updated. A new node willing to join the system first contacts a bootstrapping server and gets a partial list of existing nodes. This lets the joining nodes know in advance an entry point into the network. Something similar happens when a node leaves the system. When a node leaves the system or becomes unreachable, nodes that point to that node are affected. The routing table entries are to be updated as a result. In the case when the node does not leave the network abruptly, it notifies its neighbours about its departure, and then these nodes forward the information about the departure of this node to the other nodes in the networks. But under circumstances of unexpected leave by a node, the node leaving the network will not be able to notify to the neighbouring nodes. In such a scenario, an efficient fault tolerance and detection mechanism, such as keep-alive messages or periodic checking, is required.

Pastry was proposed in 2001 by Antony Rowstron and Peter Druschel and was developed at Microsoft Research Ltd, Rice University, Purdue University and University of Washington.

In Pastry, data items and nodes have a unique 128-bit IDs ranging from 0 to $2^{128} - 1$. For routing, these IDs are treated of as sequences of digits in base 2^b . Typically, $b = 4$, so these digits are hexadecimal. These IDs are arranged as a circle modulo 2^{128} . These node IDs are generated randomly and distributed in the ID space. These nodes are arranged as a ring, in the ascending order of the node IDs. So, the basic idea of routing involves finding the 128 bit hash for a given key, finding the node closest to the hash value and subsequently, sending the request to that particular node.

Each node in Pastry maintains 3 tables, Routing table, Leaf set and Neighbourhood set.

Routing table contains $\lceil \log_{2^b} N \rceil$ rows with 2^b number of columns, where N is the number of pastry nodes.



The entries at row i point to nodes that share the first i digits of the prefix with the key. Each cell refers to a base 2^b digit. If the digit associated with a cell matches the $(i+1)^{\text{th}}$ digit of the key, then we have a node that matches the key with a longer prefix. The request is then routed to that node. Thus, Pastry can route to the numerically closest node to a given key in less than $\text{floor}(\log_{2^b} N)$ steps under normal operation.

Leaf set is a set of L nodes that contains $L/2$ nodes with the numerically closest larger node IDs and $L/2$ nodes with numerically smaller node IDs.

Neighbourhood set M contains the node IDs and the IP addresses of the IM nodes that are closest to the local node. This set is normally not used in routing messages, but it is useful in maintaining locality properties.

Routing Algorithm:

1. Send the join request to the neighbours.
2. Maximal prefix matching of the node ID with its neighbouring node ID in the leaf set.
 1. If found in leaf set
 1. Send the join request to the neighbours
 2. Send the routing table to the neighbours
 2. Else
 1. Search in routing table based on prefix
 2. Send the routing table and join request to all the corresponding nodes in the matched prefix row.
 3. Else if not found anywhere
 1. Create an entry in neighbourhood set
 2. Send the join request to every node present in neighbourhood set.

Our Implementation of Pastry:

Functions-

1. Port <X>
creating a node which has its listening port as X.
2. Join <Node ID> <IP> <Port>
Joining a new node to the existing network that has Node ID, IP and a listening Port.
3. Put <key> <value>
Store the key value pair with matched prefix Node ID
4. Get <key>
Retrieve the key value pair with matched prefix Node ID
5. lset
Prints the leaf set of a pastry node.
6. nset
Prints the neighbourhood set of a pastry node.
7. routetable
Prints the routing table of a pastry node.
8. quit
Deletes a pastry node and moves its data to one of its neighbourhood nodes.
9. shutdown
Shuts down the entire pastry network.

Base Assumptions:

1. 4 Byte node ID
2. 6 Byte key
3. MD5 cryptographic function is used to generate Node ID and key for the network.

Structures:

```
struct DHT {  
    unordered_map<string, string> distributedHashTable;  
};
```

DistributedHashTable is a structured database where the key_ID serves as a key and uniquely identifies the value across the network. Every node ID has only those key value pairs for which its node ID's prefix matches.

```
struct routingTable {  
    vector< pair<string,int> > neighbour_set;  
    vector< pair<string,int> > leaf_set;  
    map<string,vector<pair<string,int> > > routing_set;  
};
```

Every node has a routing table that has the 3 components, namely leaf_set, routing_set and a neighbour set.

neighbour_set has entries based on proximity. neighbour_set is a vector of pairs that stores the node IP and the corresponding port number.

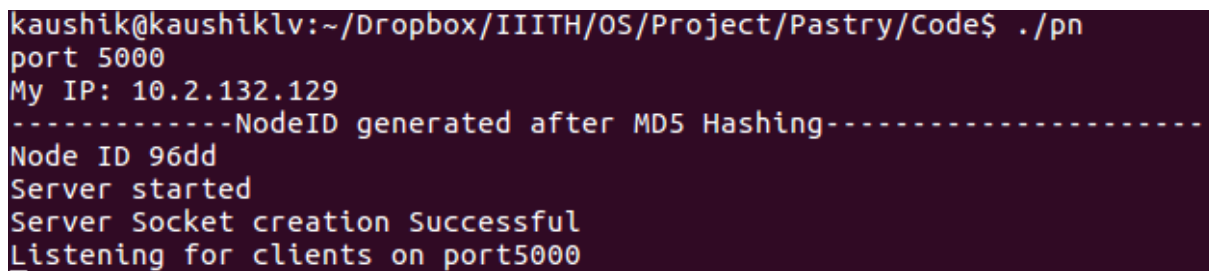
leaf_set has entries based on logical proximity. leaf_set is a vector of pairs that stores the node IP and the corresponding port number.

routing_set has entries based on overlay network of Pastry. routing_set is a map that stores pair of node IP and port at the key of map which matches with prefix of nodeID.

Function Implementation:

Port <X>

```
if(command == "port") {
    connection.port = stoi(remaining);
    temp += connection.IP;
    temp += remaining;
    nodeID = md5(temp);
    nodeID = nodeID.substr(0, 4);
    cout << "Node ID " << nodeID << endl;
}
```



A terminal window with a dark purple background. The prompt is 'kaushik@kaushiklv:~/Dropbox/IIITH/OS/Project/Pastry/Code\$'. The user enters './pn' followed by 'port 5000'. The output shows the IP '10.2.132.129', a separator line, the generated Node ID '96dd', and confirmation that the server started and is listening on port 5000.

```
kaushik@kaushiklv:~/Dropbox/IIITH/OS/Project/Pastry/Code$ ./pn
port 5000
My IP: 10.2.132.129
-----NodeID generated after MD5 Hashing-----
Node ID 96dd
Server started
Server Socket creation Successful
Listening for clients on port5000
```

Join <Node ID> <IP> <Port>

```
map<string,vector<pair<string,int> > >::iterator it;
if(join_nodeID.substr(0, join_nodeID.length() - 1) ==
    nodeID.substr(0,nodeID.length() - 1))
{
    // connect and send
    node.leaf_set.push_back();
}
else if(where != LEAF_SET){
    // searching for first 2 chars
    if(join_nodeID.substr(0, join_nodeID.length() - 2) ==
        nodeID.substr(0,nodeID.length() - 2))
    {
        where = ROUTING_TABLE;
        it = node.routing_set.find(join_nodeID.substr(0, 2));
        if(it != node.routing_set.end())
            //emp.push_back();
    }
}
```

```

        else if(join_nodeID.substr(0, join_nodeID.length() - 3) ==
                nodeID.substr(0,nodeID.length() - 3)) {
            where = ROUTING_TABLE;
            it = node.routing_set.find(join_nodeID.substr(0, 1));
            if(it != node.routing_set.end())
                //temp.push_back();
            else
                where = NEIGHBOUR_SET;
                node.neighbour_set.push_back();

        }
        if(where == LEAF_SET)
            // send to every node of leaf_set

        else if(where == ROUTING_TABLE)
            //send to every node of routing table matched key

        else if(where == NEIGHBOUR_SET) {
            // send to every node of neighbour set

        }

        connectToNetwork(join_IP, stoi(join_port), NOTHING);
        //send its own routing table to join requested node

```

```

kaushik@kaushiklv:~/Dropbox/IIITH/OS/Project/Pastry/Code$ ./pn
port 5000
My IP: 10.2.132.129
-----NodeID generated after MD5 Hashing-----
Node ID 96dd
Server started
Server Socket creation Successful
Listening for clients on port5000
join ce1a 10.2.132.129 6000

-----Details of Join Request-----
Join IP 10.2.132.129
Join Port 6000
Join NodeID ce1a

-----Connection Established with other node through Port: 6000-----
join$#$ce1a$#$10.2.132.129$#$6000$#$
-----Connection Established with other node through Port: 6000-----

```

Put <key> <value>

```
map<string,vector<pair<string,int> > >::iterator it;
    if(key_Id == nodeID)
        distributedHashTable.insert(make_pair(key_Id, value));

    else if(key_Id.substr(0, key_Id.length() - 1) ==
            nodeID.substr(0,nodeID.length() - 1)) {
        // send put request to every node of leafset
    }
    else if(where != LS) {
        if(key_Id.substr(0, key_Id.length() - 2) ==
            nodeID.substr(0,nodeID.length() - 2)) {

            it = node.routing_set.find(key_Id.substr(0, 2));
            if(it == node.routing_set.end()) {

                }
            else {
                where = RT;
            }
        }
    }
    if(where == LS) {
        // send put request to every node of leafset
    }
    else if(where == RT) {
        // send put request to every node of routing table
    }
}
else if(where == NS) {
    // send put request to every node of neighbour set
}
```

```
kaushik@kaushiklv:~/Dropbox/IIITH/OS/Project/Pastry/Code$ ./pn
port 5000
My IP: 10.2.132.129
-----NodeID generated after MD5 Hashing-----
Node ID 96dd
Server started
Server Socket creation Successful
Listening for clients on port5000
join ce1a 10.2.132.129 6000

-----Details of Join Request-----
Join IP 10.2.132.129
Join Port 6000
Join NodeID ce1a

-----Connection Established with other node through Port: 6000-----
join$#$ce1a$#$10.2.132.129$#$6000$#$

-----Connection Established with other node through Port: 6000-----
put 96ddee Krishna
-----Key-Value Pair Inserted into Distributed Hash Table-----
96dd Krishna
```


Get <key>

```
map<string,vector<pair<string,int> > >::iterator it;
    if(key_Id == nodeID) {
        distributedHashTable.find(make_pair(key_Id, value));
    }
    else if(key_Id.substr(0, key_Id.length() - 1) ==
            nodeID.substr(0,nodeID.length() - 1)) {
        // send Get request to every node of leafset
    }
    else if(where != LS) {
        if(key_Id.substr(0, key_Id.length() - 2) ==
            nodeID.substr(0,nodeID.length() - 2)) {

            it = node.routing_set.find(key_Id.substr(0, 2));
            if(it == node.routing_set.end()) {

                }
                else {
                    where = RT;
                }
            }
        if(where == LS) {
            // send Get request to every node of leafset

        }
        else if(where == RT) {
            // send Get request to every node of routing table

        }
        }
        else if(where == NS) {
            // send Get request to every node of neighbour set

        }
    }
```

quit

```
pair<string, int> neighbour;
    if(!node.neighbour_set.empty()) {
        neighbour.first = node.neighbour_set[0].first;
        neighbour.second = node.neighbour_set[0].second;
        // Send its distributed hash table to first node of neighbour set.

        for(auto i = node.neighbour_set.begin(); i !=
            node.neighbour_set.end(); i++) {
            // Inform other nodes in neighbourhood set of its
                non-existence.
        }
    }
}
```

shutdown

```
for(every node in routing table) {
    // Inform all the nodes in entire pastry network of shutdown.
}
```