SOFTWARE DESIGN DOCUMENT

DISTRIBUTED LEDGER

Submitted By:

**PRATIBHA (2017MCS2087)**

**PRIYA (2017MCS2095)**

1. **Introduction**

**Purpose**

The aim is to design a distributed ledger system , which uses peer to peer technology. In the project ,we have created network of N nodes. Each node has unique id which is randomly generated using random probability , and a file called Ledger having entries- transaction id, sender id, receiver id , witness id , amount . Whenever a transaction commits ,this file is updated and copy of this file sent to every live node. Whenever any transaction happen ,it is known to every node in network, so it is easy to validate any transaction. All node will have same copy of ledger. We assumes that there is [stable storage](https://en.wikipedia.org/wiki/Stable_storage) at each node with write -ahead , that no node crashes forever, that the data in the write-ahead log is never lost or corrupted in a crash, and that any two nodes can communicate with each other.

The aim is to is to implement the file sharing system among large no of node connected in p2p network. It is scalable, open distributed which is called **Distributed Hash Table**. DHT is distributed hash table. It distributes the all data in p2p network and can also retrieve data from node efficiently. For this it uses the hashing. All data are distributed among the node using hashing. Each node having unique id which associated with unique key value. Data are stored in DHT by hashing it with appropriate hash function which produces key value and these key values are associated with some node. When this hashed key value mapped with node’s key value that data go and stored at that node and in similar we can retrieve the data from node.

1. **SYSTEM REQUIREMENT SPECIFICATION**
   1. **Software Requirements**

* Operating System : Windows XP, 7,Ubuntu Linux 10 -17.04.
* Tool Used : Mininet.
* Programming Language used : Python

**2.2 Hardware Requirements**

* Processor : Pentium III at least
* CPU Clock : 700MHZ or more
* Main Memory : 128MB RAM or more
* Hard Disk Drive : 20GB
* Cache Memory : 512K
* Virtual Memory : at least 250MB
* Monitor : 15 inch COLOR MONITOR/laptop

1. **Work Done**

**3.1 Mininet**

Mininet is a network emulator which creates a network of virtual hosts, switches, controllers, and links. Mininet provides an easy way to get correct system behaviour and to experiment with topologies.  It uses lightweight virtualization to make a single system look like a complete network, running the same kernel, system, and user code. A Mininet host behaves just like a real machine and run arbitrary programs .

* Mininet set up command :

1. sudo apt-get install mininet
2. sudo mn -c
3. sudo apt-get install git
4. git clone git://github.com/mininet/
5. cd mininet
6. git tag #list available versions
7. git checkout-b cs244-spring-2012-final
8. mininet/util/install.sh -a
9. sudo wireshark &
10. sudo mn
    1. **. CREATING TOPOLOGY**

A simple network topology (based on mininet/topo.py:SingleSwitchTopo) which consists of a N number of hosts (h1 through hN) connected to a single switch (s1):

class SingleSwitchTopo(Topo):

"Single switch connected to n hosts."

def \_\_init\_\_(self, n=2, \*\*opts):

# Initialize topology and default options

Topo.\_\_init\_\_(self, \*\*opts)

switch = self.addSwitch('s1')

# Python's range(N) generates 0..N-1

for h in range(n):

host = self.addHost('h%s' % (h + 1))

self.addLink(host, switch)

* **Topo()** : the base class for Mininet topologies
* **build():** The method to override in your topology class.
* **addSwitch():** adds a switch to a topology and returns the switch name
* **addHost():** adds a host to a topology and returns the host name
* **addLink(**): adds a bidirectional link to a topology.
* **Mininet**: main class to create and manage a network
* **start()**: starts your network
* **stop():** stops your network
  1. **CREATING PEERTO PEER NETWORK**

A peer to peer network is the network in which there is no central server . all the peers act as server and client both. This is done by SimpleServer.py and ClientRequest.py file.

* SimpleServer.py – this create sever socket that is present in passive mode and waiting for client.
* ClientRequest.py- this create client socket that is present in active mode .
* Initially all the nodes run as the server.
* A server socket is created on all nodes using

s= socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

* Socket is bind to particular IP address and port of the server node.

s.bind(Server IP,Server Port)

* All the nodes are listening and waiting for some peer

s.listen(5)

* When a peer wants to do any transaction , it sends a client request to get a connection to server IP and Server Port.

Clientsock=socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

Clientsock.connect(SERVER IP,SERVER Port).

**3.4 . Implement DHT**

Chord is protocol used to implement peer to peer Distributed Hash Table.  A DHT store key, value pair  by assigning keys to different computers nodes, a node stores the values for certain keys for which it is responsible. 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.

**Working**

The Chord protocol supports just one operation: given a key, it will determine the node responsible for storing the key's value. Each node are acting as both server and client, has a unique 160-bit node identifier, produce by appropriate hash function, by hashing the node's IP address. the IDs as occupying a circular identifier space. Hash keys are also mapped into this identifier space, by hashing them to 160-bit key identifiers. Chord defines the node responsible for a key to be that key's ``successor.'' The successor of a key or node identifier *j* is the node with the smallest ID that is greater than or equal to *j*. Chord's primary task is to find these successors.

Chord protocol needs to perform three operations:

* initialize the predecessor and fingers;
* update the fingers and predecessors of existing nodes to reflect the change in the network topology caused by the addition of *n*; and
* copy all keys for which node *n* has became their successor to *n*.

**Finger Table**

Each node, *n*, maintains a routing table with 160 entries, called the *finger table*. The *i*th entry in the table at node *n* contains the identity of the *first* node, *s*, that succeeds *n* by at least 2^(i-1) on the identifier circle, i.e., *s* = successor(n + 2^(i - 1)), where 1 ≤ *i* ≤ 160. The node *s* is called the *i*th *finger* of node *n*, and denoted by *n.finger[i].node*. The first finger of *n* is its immediate successor on the circle. In addition, the *i*th finger table entry of node *n* contains the interval, *[n.finger[i].node, n.finger[i+1].node)*, called the *i*th *finger interval* of node *n*, and denoted by *n.finger[i].interval*.

When a node *n* does not know the successor of a key *k*, it sends a ``find successor'' request to a intermediate node whose ID is closer to *k*. Node *n* finds the intermediate node by searching its finger table for the closest finger *f* preceding *k*, and sends the find successor request to *f*. Node *f* looks in its finger table for the closest entry preceding *k*, and sends that back to *n*. As a result *n* learns about nodes closer and closer to the target ID.

.