Simulation of Chord Peer-to-peer Network: Distributed System (Graduate Course)

Chih-Yuan Lin
chih-yuan.lin@liu.se
Department of Computer and Information Science
Linköping University

September 3, 2018

Introduction

This report summarizes the project work of the Distributed System course. Our assignment was to implement a basic simulator of Chord peer-to-peer network proposed by Stoica et al [1, 2] and measure the overhead of node insertion and message forwarding scales with the number of nodes. The report first introduces the Chord algorithm in the first section and then describes our implementation in the second section. Finally, it shows the settings and results of experiments and compares the results with the proposed performance in the original papers.

Background

Chord is a scalable peer-to-peer lookup service for Internet applications. Each Chord node has a unique m-bit identifier (ID) locating in a virtual ID cycle. Given a key, Chord defines the node which is responsible for the key to be the successor of the key's ID. The successor of an ID j is the node with the smallest ID that is greater than or equal to j.

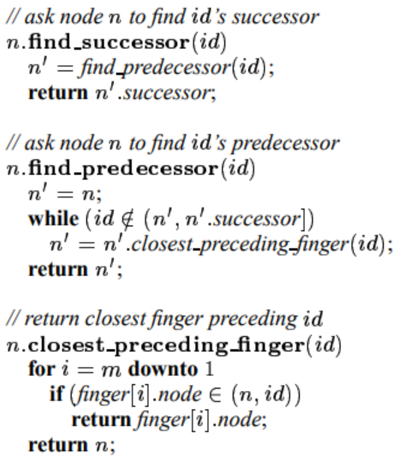
A Chord node needs two data structures to perform lookups: a successor list and a finger table. Lookups performed only with successor list would require an average $N/2$ message exchanges, where N is the number of nodes. To reduce the number of message exchanges to average $\frac{1}{2}logN$, each node remains a finger table with m entries. The i^{th} entry in the table at node n contains the ID of the first node that succeeds n by at least 2^{i-1} on the circular space. Fig. 1 shows pseudo-code to complete a lookup operation. If a key's ID falls between n and its successor, node n returns its successor. Otherwise, node n searches its finger table for the node n' whose ID most closely precedes the key's ID and then ask n' to find the key's successor.

Fig. 2 shows pseudo-code to node insertion. When a new node n joins the network, Chord must perform

1. initialize the predecessor and finger table of node n ($init\_finger\_table$).
2. Update the finger tables and predecessors of existing nodes to reflect the insertion of n ($update\_others$).

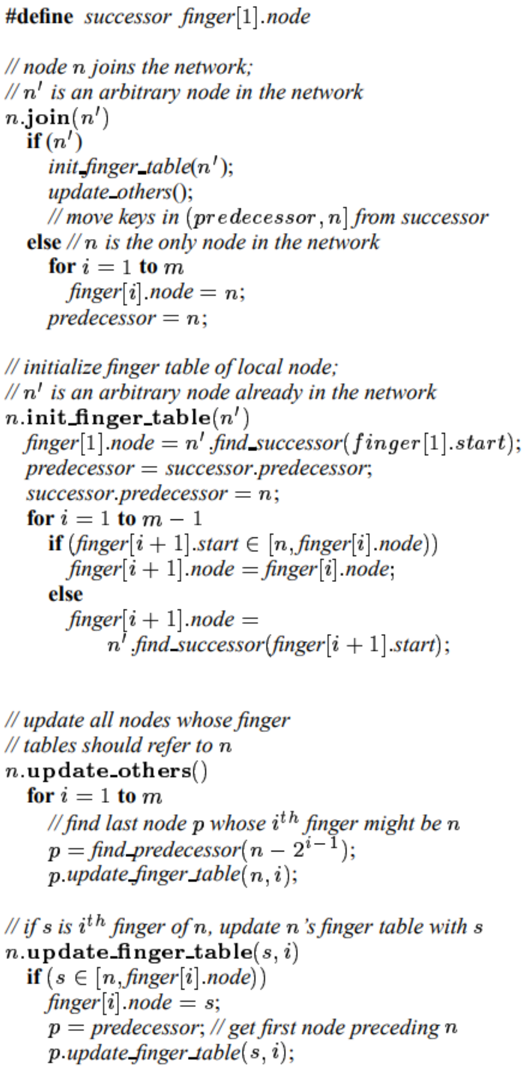
In function $init\_finger\_table$: Node n checks whether the i^{th} fingers is also the correct $(i+1)^{th}$ finger for each i. If not, n learns the finger value by asking n' to look it up. That is, the total number of fingers that must be looked up to $O(logN)$, where induces the overall time to $O(log^2N)$.

In function $update\_others$: For a give n, the algorithm finds node p whose i^{th} finger might be n counter-clock-wisely. The number of nodes that need to be updated is proposed as $O(logN)$ with high probability. Finding and updating these nodes takes overall $O(log^2 N)$.



```
// ask node $n$ to find $id$'s successor
$n$.find_successor($id$)
  $n' = find\_predecessor(id)$;
  return $n'.successor$;

// ask node $n$ to find $id$'s predecessor
$n$.find_predecessor($id$)
  $n' = n$;
  while ($id \notin (n', n'.successor]$)
    $n' = n'.closest\_preceding\_finger(id)$;
  return $n'$;

// return closest finger preceding $id$
$n$.closest_preceding_finger($id$)
  for $i = m$ downto $1$
    if ($finger[i].node \in (n, id)$)
      return $finger[i].node$;
  return $n$;
```

Figure 1: The proposed lookup algorithm.



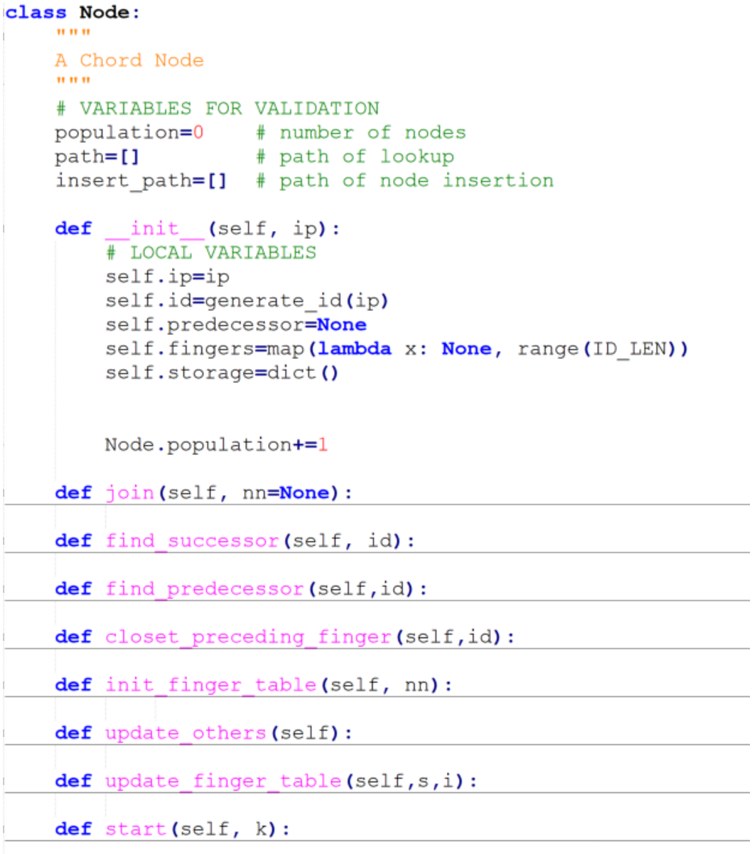
```
#define successor finger[1].node

// node $n$ joins the network;
// $n'$ is an arbitrary node in the network
$n$.join($n'$)
  if ($n'$)
    init_finger_table($n'$);
    update_others();
    // move keys in $(predecessor, n]$ from successor
  else // $n$ is the only node in the network
    for $i = 1$ to $m$
      $finger[i].node = n$;
    $predecessor = n$;

// initialize finger table of local node;
// $n'$ is an arbitrary node already in the network
$n$.init_finger_table($n'$)
  $finger[1].node = n'.find\_successor(finger[1].start)$;
  $predecessor = successor.predecessor$;
  $successor.predecessor = n$;
  for $i = 1$ to $m - 1$
    if ($finger[i+1].start \in [n, finger[i].node)$)
      $finger[i+1].node = finger[i].node$;
    else
      $finger[i+1].node =$
          $n'.find\_successor(finger[i+1].start)$;


// update all nodes whose finger
// tables should refer to $n$
$n$.update_others()
  for $i = 1$ to $m$
    // find last node $p$ whose $i^{th}$ finger might be $n$
    $p = find\_predecessor(n - 2^{i-1})$;
    $p.update\_finger\_table(n, i)$;

// if $s$ is $i^{th}$ finger of $n$, update $n$'s finger table with $s$
$n$.update_finger_table($s, i$)
  if ($s \in [n, finger[i].node)$)
    $finger[i].node = s$;
    $p = predecessor$; // get first node preceding $n$
    $p.update\_finger\_table(s, i)$;
```

Figure 2: The proposed node insertion algorithm.

Implementation

Our simulator was implemented with Python in an object-oriented manner. A Chord network is maintained by a list of Node instances. Messages are forwarded as arguments between public methods of two instances (i.e, without passing through RPC). Fig. 3 shows the class variables and methods.

In this project, we implement a simple Chord-based lookup service. Each Chord node has a variable *ip* keeping its IP address and an *ID_LEN*-long *id* which is produced by hashing the IP address with SHA-1 function. To improve the scalability, Chord protocol requires more routing information than just a successor pointer. Variable *predecessor* is the predecessor pointer and *fingers* is the finger table. The finger table contains *ID_LEN* entries. The i^{th} entry of node n stores the pointer of the first node, s, that succeeds n by at least 2^i, where $1 \leq i \leq m$. Last, we implement the storage with the build-in dictionary data type, that stores pairs of strings and secret numbers.



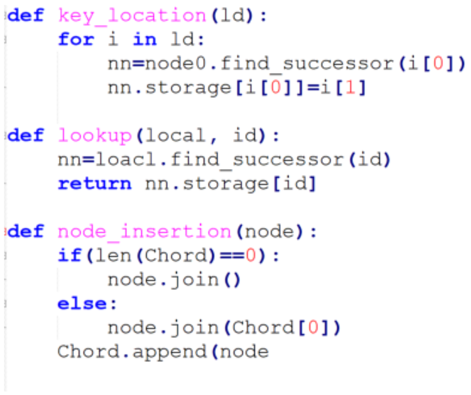
```
class Node:
    """
    A Chord Node
    """
    # VARIABLES FOR VALIDATION
    population=0     # number of nodes
    path=[]          # path of lookup
    insert_path=[]   # path of node insertion

    def __init__(self, ip):
        # LOCAL VARIABLES
        self.ip=ip
        self.id=generate_id(ip)
        self.predecessor=None
        self.fingers=map(lambda x: None, range(ID_LEN))
        self.storage=dict()


        Node.population+=1

    def join(self, nn=None):

    def find_successor(self, id):

    def find_predecessor(self,id):

    def closet_preceding_finger(self,id):

    def init_finger_table(self, nn):

    def update_others(self):

    def update_finger_table(self,s,i):

    def start(self, k):
```

Figure 3: The Node class.

We also provide three APIs to manipulate the Chord network as shown in Fig. 4

1. Key location
2. Lookup
3. Node insertion

We test only the Lookup and Node insertion as required.



```
def key_location(ld):
    for i in ld:
        nn=node0.find_successor(i[0])
        nn.storage[i[0]]=i[1]

def lookup(local, id):
    nn=loacl.find_successor(id)
    return nn.storage[id]

def node_insertion(node):
    if(len(Chord)==0):
        node.join()
    else:
        node.join(Chord[0])
    Chord.append(node
```

Figure 4: The defined APIs

Performance

This section describes how we measure the overhead of node insertion and message forwarding and the resulting performance.

Experiment Settings

Message forwarding. Message forwarding in a Chord network is triggered by a lookup operation. The original paper evaluates the lookup performance with the metric path length, which is defined as the number of nodes traversed during a lookup operation. The paper presents the path length in a network that contains 2^k random nodes and 100×2^k random keys in all. It varies the k from 3 to 14 and conducts an separate experiment for each k. Each node in each experiment picked a random set of keys to query from the system, and it measures the path length required to resolve each query on average. In order to compare our work with the original paper, we follow the same settings of experiments.

Node insertion. Node insertion performance is not tested in the original paper. Thus, we extend the definition of path length here. We measure the number of nodes traversed during *init_finger_table* and *update_finger_table* and refer to this measurement as path length of node insertion. Our experiments to measure the overhead of node insertion are conducted in the same Chord networks as the above. For each experiment, we insert a random node into a 2^k random network. We repeats the experiment 20 times for each k.

Results

Fig. 5 presents the median, 1^{st}, and 99^{th} percentiles of path length of lookup as a function of k. The red line is generated by $\frac{1}{2}logN$ as mentioned in the background section. We can see that the path length increases logarithmically as proposed. Compare to the proposed performance in Fig. 6, the length of routing path is quite similar.

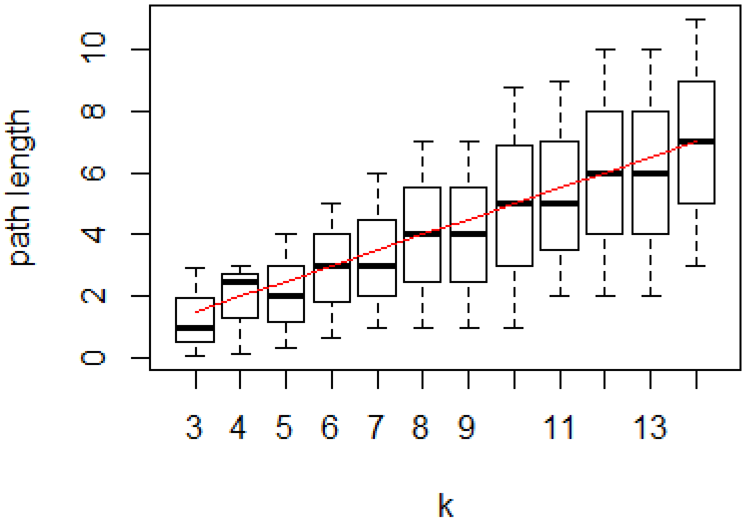


Figure 5: The path length of lookup operation as a function of k.

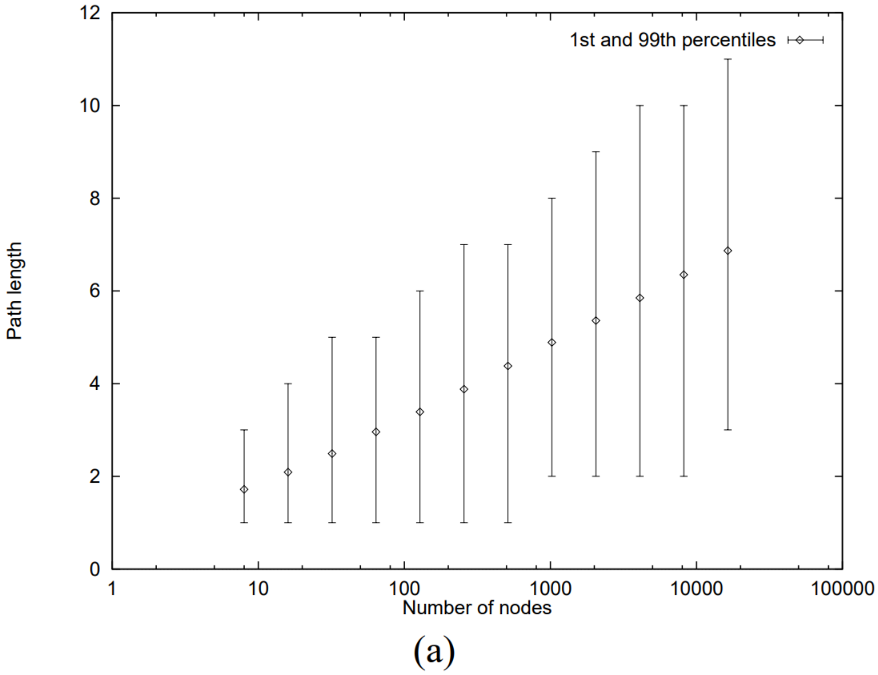


Figure 6: The path length of lookup operation as a function of network size.

Fig.7 shows the median, 1^{st}, and 99^{th} percentiles of path length of node insertion as a function of k. The red line is generated by $200 + 4.5log^2N$, where the coefficients are determined experimentally. The concludes the performance of insertion is $O(log^2N)$ as mentioned in the background section.

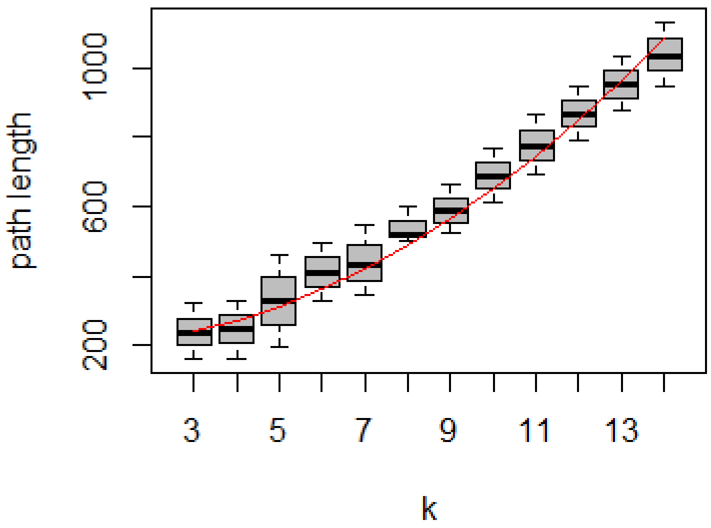


Figure 7: The path length of node insertion as a function of *k*

References

[1] Ion Stoica, Robert Morris, Davide Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the SIGCOMM'01*, 2001.

[2] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. Technical report, TR-819,MIT LCS, 2001.