In-memory Data Grid on Improved Chord

CSE 534 Fundamentals of Computer Networks

Project by

Naveen Gaddam [111344916]
Shalaka Sidmul [111367731]

Introduction

Chord has been widely used as a routing protocol in structured peer-to-peer overlay networks. A fundamental problem of peer-to-peer applications is to efficiently locate the node that stores a particular data item. Chord, a distributed lookup protocol that addresses this problem. The performance of structured peer-to-peer overlay networks depends on the routing protocols. The original algorithm however, has some redundancies which can be reduced to further improve performance. The goal of this project is to present an improvement strategy over the original Chord routing algorithm. We demonstrate this improvement by running an in-memory Distributed Data Grid over Chord protocol.

THE CHORD ALGORITHM

The Chord protocol [1] supports just one operation: given a key, it maps the key onto a node. Depending on the application using Chord, that node might be responsible for storing a value associated with the key. Chord uses consistent hashing [2] to assign keys to Chord nodes. Consistent hashing tends to balance load, since each node receives roughly the same number of keys, and requires relatively little movement of keys when nodes join and leave the system.

The key for data that is to be stored, is hashed using a consistent hashing function to obtain its identifier in the key space, and assigned a node. The selected node is one whose identifier is a successor of the identifier of the key. Chord uses a flat key space. Meaning, nodes and keys are given an m-bit identifier in the same key space.

Consistent hashing assigns keys to nodes as follows. Identifiers are ordered on an identifier circle modulo 2^m. Key 'k' is assigned to the first node whose identifier is equal to or follows (the identifier of) in the identifier space.[1]

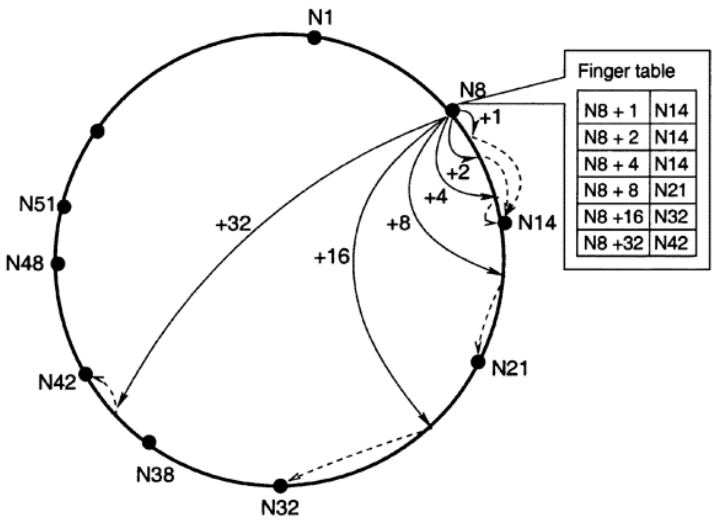
The Chord Finger Table

To search for a key in the Chord ring the naïve way is to perform a linear search on the nodes in the ring. This is however, inefficient. To avoid the linear search, Chord implements a faster search method by requiring each node to keep a *finger table* containing up to *m* entries, where *m* is the number of bits in the hash key. . A Finger table is like the routing table of network layer routers. It consists of key-value pairs, called fingers, where keys and values are derived using as follows:

Key: (node_id) + 2^{i-1}

Value: successor(Key).

Consider the following example:



The figure alongside shows the finger table of node N8 in the Chord ring.

A lookup for, say, N8 + 12, will return node N32 as per the Finger Table.

Figure: Original Chord Ring and Finger Table

However, as can be seen from the Finger Table, the entry N14 is repeated thrice. The constraint of having *m* entries in the Finger Table limits the coverage of the key space as there are redundant entries. As seen from the figure above, the number of the intermediate nodes between the source node sending out the lookup request and the largest bound node which it can points to is half the total number of the remaining nodes except for the two nodes. In the figure, the largest bound node of N8 is N42 and the number of the intermediate nodes is 4. Thus, lookup of a key stored beyond N42 is not possible directly from the Finger Table of node N8. This increase in the number of hops to take in order to locate the successor of a key beyond N42 can be avoided. In this report, we present an optimization strategy for reducing this redundancy and improving lookup performance of Chord.

PROBLEM STATEMENT

The Chord algorithm requires each node to periodically run a function '*fix_fingers()*' to make sure it's finger table entries are correct and reflect the current state of the ring because the peer nodes may leave or join the ring at any time.

```
n.fix_fingers()
  next = next + 1; /* the next finger to fix */
  if (next > m)
    next = 1;
  finger[next] = find_successor(n+ 2^(next-1));
```

As evident from the method above, the finger table entries are updated without taking duplicate values into consideration. This introduces the redundancy and limits coverage of the key space to not more than half the Chord ring. This results in more number of hops to be taken for looking up a key in the ring. Also, the time taken for the lookup is directly proportional to the number of hops.

APPROACH

The algorithm for *fix_fingers* can be modified to take duplicate values into consideration. Thus every node will have a finger table with unique fingers. With this we try to increase the coverage of the ring by each node's finger table, making it denser and enlarge the lookup bound.

The node identifiers are a function of the IP addresses of the nodes. Thus, all nodes may not be equidistant from each other on the Chord ring. Also, having exactly *m* fingers on all nodes at all times is unnecessary as number of nodes can change dynamically.

Our approach is based on the research paper "An Improvement to the Chord-based P2P Routing Algorithm" [3]. We analyze the improvement suggested in the paper and further enhance it by improving the algorithm for fix_fingers().

The improvement strategy suggested in the paper is as follows:

1. Initialize counter to zero.
2. For each finger:
    a. If the node id in the finger is same as the previous one, increment counter.
3. If counter > zero,
    a. For I from 1 to counter + 1, calculate new finger as follows:
        i. Key: [k+i*(2m -1+n-m)/(count +1)]
        ii. Value: successor(Key)

This strategy improves the finger table, making it denser and eliminating duplicate entries.

Our approach is to combine the two step process of finding duplicates and then replacing them, into a single consolidated step.

As and when a finger is selected to be fixed by *fix_fingers()*, before adding the finger into the finger table, the value is checked for being a duplicate.

SOLUTION

We modified the fix_fingers() method as follows:

```
for(i = 2 ; i< 33; i++){
    id =  (long) ((localId + Math.pow(2, i - 1)) % Math.pow(2, 32));
    succ = localNode.findSuccessor(id,null);
    InetSocketAddress currEntry = this.fingerTable.getEntry(i);
    if(currEntry == null){
        /* key is not present */
        if (this.fingerTable.isUnique(i,succ)){
            this.fingerTable.update("UPDATE_FINGER", i, succ);
        }
    }else{
        if (this.fingerTable.isUnique(i,succ)){
            this.fingerTable.update("UPDATE_FINGER", i, succ);
        }else{
            this.fingerTable.update("REMOVE_ENTRY", i, succ);
        }
    }
}
```

Explanation:

Finger table : Map of Entry number versus Address of successor node.

The first entry in the finger table is the immediate successor node.

This entry is periodically fixed by *stabilize()*. Hence, *fix_fingers()* deals with all the remaining entries. Thus, on each node in the Chord ring, for each key obtained from:

$(node\_id) + 2^{i-1}$ where $1 < i < 33$

Find the successor of the key and update finger table as follows:

1. If the key being updated is not present in the finger table:
   a. If the successor address obtained is unique among all previous entries in the finger table
      i. Add the entry to the finger table

2. If key being updated is present in the finger table,
    a. If the successor obtained is unique among all previous entries
        i. Add the entry to the finger table
    b. Else, remove the entry for the key in question from the finger table.

Our approach not only makes the finger table denser, but also eliminates the constraint of having *m* entries in the finger table. Thus, every node will have only unique fingers and the size of finger table for each node is dependent on the position of the node in the ring. This does away with the fact that nodes may not be equidistant from each other on the key space of Chord ring.

To demonstrate the improvement, we hosted an **in-memory data grid** on Chord nodes. Our application implements the client-server paradigm. The client can perform following operations:

1. Save data on the ring
2. Retrieve stored data

Our experimental evaluation measures the time taken as well as the number of hops taken for resolving the client's request.

Assumptions

1. Effects of other factors such as bandwidth, load balance, and network traffic are consolidated into a delay introduced between request-response exchanges between client node and Chord node as well as among Chord nodes.
2. We assume the bandwidth available in original chord and improved chord setup to be same.
3. As the data is stored in-memory and not on a database, the time taken to establish a connection with database and query for the request on the database is not evaluated.

Evaluation Setup

1. **Tools**: Mininet is used to simulate the structured peer-to-peer network.
2. **Technologies**: Python and Quagga to build the network, Java and Socket programming to run the Chord algorithm.
3. **Hashing Algorithm**: Used SHA1 hashing algorithm in Chord to maintain a Hash Map at each node. This hashing algorithm is used in storage and lookup.

4. **Network Simulation**: We have virtualized the network topology using Mininet and ran Chord over the topology. We have setup static routes between the nodes.
5. **Parameters evaluated**: The experiment task is to compare the original Chord protocol algorithm with our improved algorithm. For this purpose,
    1. We compared the average path length and average time taken for lookup for original Chord and improved Chord.
    2. We ran the tests for 5, 6, 7, 8, 10, 15, 20 and 30 number of nodes in each topology.
    3. For each test, we did a lookup for 30 requests and 116 requests, and plotted graphs for the average of these values.
6. **Environment**: Experiments are done under the development environment of a desktop computer with 4GB memory, 2.6 GHz Intel CPU with Windows 10 Operating System.

Our Hypothesis

By eliminating the redundant entries in the finger table and adding only unique entries which covers some nodes in the second half of the ring as well, we reduce number of lookups at each node to reach the destination successor node. This in turn reduces the lookup path and lookup mean time.

Steps to Run

1. Navigate to the folder which contains start.py in terminal and run the command: python start.py
2. Run the script to run Chord on each node:
   Original Chord: OrgChord.sh
   Improved Chord: ImpChord.sh
3. Start Client: Navigate to the folder (/src/main/java/fcn) and run the command java project.chord.node.client.ClientNode
4. Command to start a node: Navigate to the folder (/src/main/java/fcn) and run the command java project.chord.Main <IP1> <IP2>, where IP1 is the node IP address and IP2 is the IP address of the contact node. We have fixed the port number for all the nodes to 9999.
5. To save data on a Node, we need to provide key-value data and IP of the node to which we send the request.

Evaluations

First, we compare the lookup_mean of the original Chord protocol with the lookup_mean of the improved Chord protocol using the new Finger Table organization structure. Fig. 1 shows that the improved Chord effectively can reduce the delay time of lookup and improve the efficiency of the lookup algorithm.

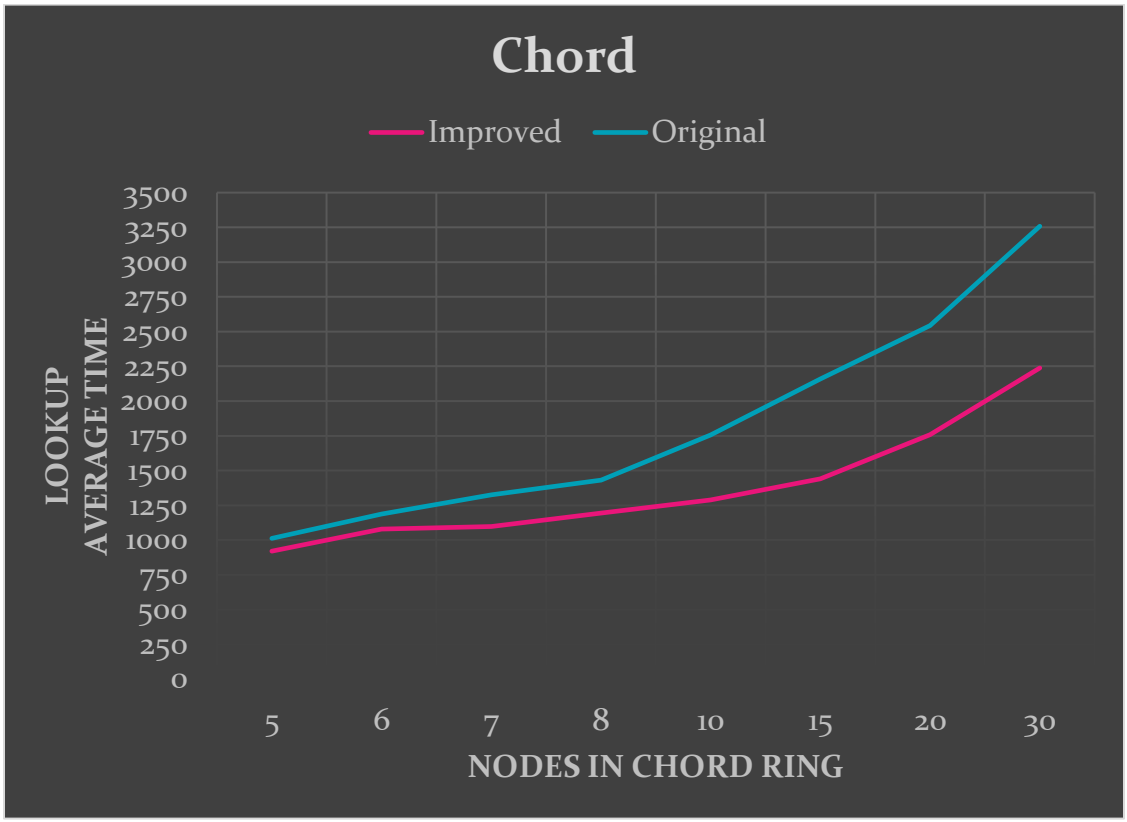


Fig. 1 Lookup average time vs #nodes

Then in the same simulation environment, we compare the average lookup path length. The lookup path length is defined as the number of nodes visited by the lookup for a distributed lookup task. From Fig.2, the path length becomes significantly shorter, so the performance of the Chord routing algorithm is improved.

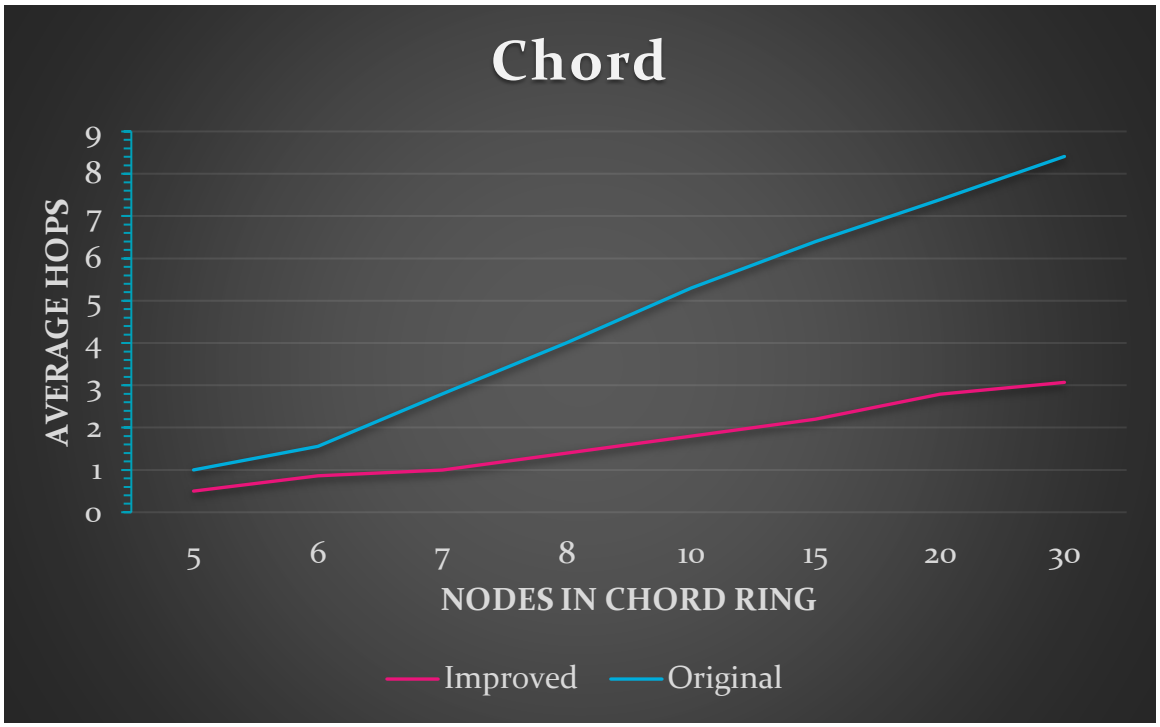


Fig. 2 Lookup average hops vs #nodes

Observations:

Minimum Chord path length is improved from O(log(n)) in original Chord to < O(log(n)) in improved Chord.

Conclusion:

In the original Chord model, Finger Table consists of redundant entries. We present an improvement to Finger Table structure to enhance the routing performance of Chord-based P2P networks. As we can see, the performance of the improved Chord is much better than the original Chord in the above graphs.

The physical layer and the Chord overlay network are linked through the Hash function. However, the construction of the overlay network does not take full advantage of the physical layer information. It is possible that on the physical layer two peers are neighbour nodes, but on the overlay network they are in a long distance. In addition, our improvement strategy only supports single keyword lookup.

GitHub Repository: https://github.com/shalakhansidmul/main

Note:

Approach and *Evaluations* are marked as the remaining 30% weightage mentioned in the project grading.

References:

[1] https://pdos.csail.mit.edu/papers/ton:chord/paper-ton.pdf

[2] D. R. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web," in Proc. 29th Annu. ACM Symp. Theory of Computing, El Paso, TX, May 1997, pp. 654–663.

[3] https://pdfs.semanticscholar.org/1172/10d8ba2162baab2fcf0a862b3ded5f80fe87.pdf

[4] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany, pp. 329-350, November 12-16, 2001

[5] https://www.slideshare.net/GertThijs/chord-presentation