The Third Mandatory Programming Assignment

Chord Protocol

IN5020/IN9020 Autumn 2025

Objective

- To develop a P2P protocol and analyze its performance
- Developing routing table for Chord protocol
- Developing look up mechanism for Chord protocol
- Developing overlay network for Chord protocol

Scope

The core operation in most peer-to-peer systems is the efficient location of data items. Given a key, the P2P protocol maps the key to one of the nodes that store the corresponding data item. Chord protocol is one such protocols that provides an efficient mechanism to perform look-up operations.

For this assignment you have to implement the following components of Chord protocol: a) construct overlay network, b) build Routing Table and b) implement Look Up algorithm. In Chord protocol, the overlay network consists of ring topology where the nodes are organized based on the node identifier. Each node maintains a routing table called "Finger Table" that maintains the routing information of a set of other peers in the network. The Look Up algorithm uses the Finger Table to identify the node that is responsible for the key. Detailed information will be provided in the technical features section.

You will be provided with a minimal P2P Simulator (built in-house) to do the implementation. The Simulator contains a basic peer-to-peer network consisting of the 'N' number of nodes. The Simulator is equipped with a ChordProtocolSimulator that sets up the basic network and assigns keys to nodes. You have to do the implementation in the ChordProtocol class (protocol/ChordProtocol.java).

Technical features

The synopsis for running the experiment is as follows:

```
Java Simulator <node count> <m>
```

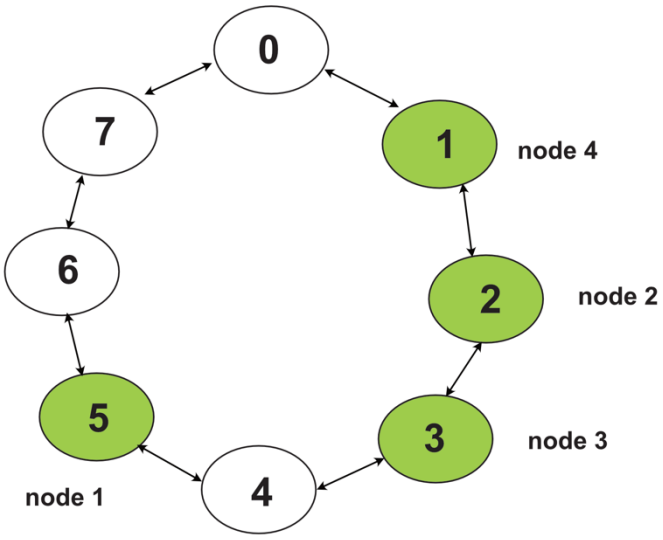
Simulator – It is the starting point of execution. The main() function in this Class will be starting the simulation and sets the protocol.

<node count> - It represents the number of nodes in the network. This value is used to create the basic network with the specified number of nodes. The Chord Protocol builds the Ring overlay network on top of the basic network.

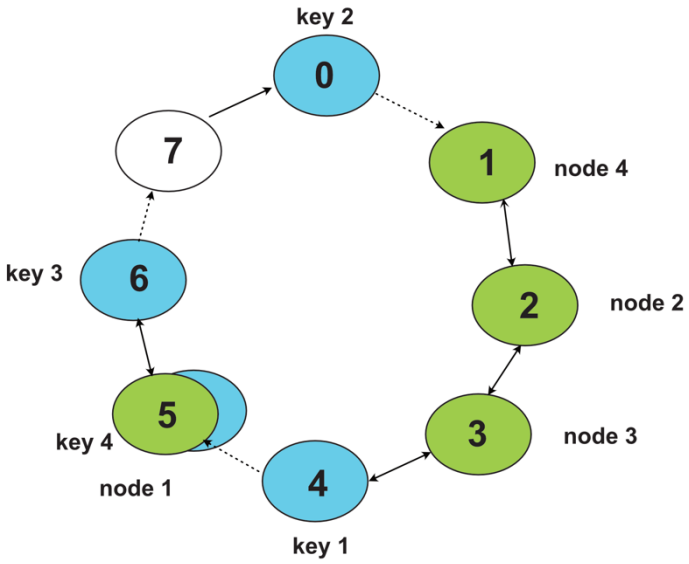
<m> - It represents the length of identifiers (m-bit) used in the Chord Protocol. For example: if m is 3 then the identifier value can range from 0 to 7.

1 Overlay Network

The overlay network consists of nodes logically organized in a ring topology. The indexes for the nodes are generated using a variant of a hash function called “consistent hashing”. (The hash function is provided in the Simulator. The ChordProtocol Class contains the object of ConsistentHashing named “ch”. The hash function can be calculated by invoking ch.hash(<data>) method). The consistent hashing returns an m-bit integer. Each node’s index is calculated by calling the hash function with the argument unique to the node. The simulator generates a unique node name and it should be used as an argument to the hash function (network.getTopology() function can be used to read the node names).
The Overlay Network consists of nodes with indexes generated using a consistent hashing function. The Overlay Network forms the logical ring where the placement of the node in the ring is determined by its index value.
Eg;- assume the node count is 4 and m is 3. The indexes for the nodes are generated as follows (node name, index): {(Node 1, 5), (Node 2, 2), (Node 3, 3), (Node 4, 1)}. The logical organization is given in the figure below.



Once the node indexes are generated, **the ChordProtocolSimulator will generate key indexes for the different keys and assigns them to the nodes based on the index**. **You don’t need to do the key assignment**. For each key, the index of the node should be greater than or equal to the index of the key and the key index should be close to the node index in the ring. For example, the key indexes are generated as follows (key name, key index): {(key 1, 4), (key 2, 0), (key 3, 6), (key 4, 5)}. Key 1 will be assigned to Node 1 since the index of the key 1 is 4 and the index of Node 1 is 5. Node 1 is situated next to key 1. Key 2 will be assigned to Node 4. Key 3 will be assigned to Node 4. Key 4 will be assigned to Node 1.



2 Routing Table

Each node maintains a routing table. The Chord Protocol uses the routing called "Finger Table" to store the routing information. The Finger Table consists of 'm' entries. The 'i'th entry (1 <= i <=m) in the Finger Table points to the information of the node (successor) that succeeds the node by at least 2^{i-1} in the ring. Each 'i'th entry consists of

1. start value. It is calculated as
   $(n + 2^{i-1}) \ mod \ 2^m \ (1 <= i <= m)$
   where n is the current node's index.
2. Interval.
   $(start, finger(i+1).start - 1)$.
   It contains the values between the range of the current start value and the next start value (if it is the last entry then the next start value should be the first entry in the table).
3. the 'i'th successor node (known as the 'i' the finger of the node) is the node that is responsible for the keys in the interval (described in step 2). The successor itself might contain the key or it might be able to route to the nodes that are responsible for the key. It should contain some identifying information about the node to enable routing to the node.

Example:-
1) For Node 3,

start	interval	successor node
$(3+2^0) \bmod 2^3 = 4$	(4,4)	Node 1
$(3+2^1) \bmod 2^3 = 5$	(5,6)	Node 1
$(3+2^2) \bmod 2^3 = 7$	(7,3)	Node 4

2) For Node 4,

start	interval	successor node
$(1+2^0) \bmod 2^3 = 2$	(2,2)	Node 2
$(1+2^1) \bmod 2^3 = 3$	(3,4)	Node 3
$(1+2^2) \bmod 2^3 = 5$	(5,1)	Node 1

The Finger Table has 2 characteristics. The first one is each node stores only information about a smaller number of nodes. The second characteristic is that each node knows more about the nodes that are placed closely in the ring than the nodes situated far away from it in the ring.

3 Look Up algorithm

The Look Up algorithm finds the node that is responsible for the given key. You can start with any node (**but the same node MUST be used for all the key lookups as the starting node**). If the current node contains the key then the Look Up operation should return the current node's information. If the current node doesn't contain the key, then the Finger Table of the current node should be checked and the correct successor should be identified. The correct successor should be the one whose interval values contain the index of the key. Once the successor is identified then the successor node's finger table should be checked. This process should be continued till the node that contains the key has been identified.
For example,

1) look up for key 1: assume that starting node is Node 4.
   - Index(Node 4) = 1.
   - Index(key 1) = 4.
   - Node 4 will check it's finger table and chooses Node 3 since Node 3 has the interval (3,4).
   - Then Node 3's finger table should be checked. Node 3 chooses Node 1 since Node 1 has an interval of (4,4).
   - Node 1 should be checked, and Node 1 contains the key. hop count is 3 (number of node's finger table has been checked including the starting node).

The keyIndexes object would be allocated with a set of keys and key indexes. You have to perform look up for each key given in the keyIndexes.
The lookup should return the following information for each key:
<key name>:<key index> <node name>:<node index> hop count:<hop count> route:<node names of those whose finger table has been checked.>
eg:- key 1:4 Node 1: 4 hop count: 3, route: Node 4 Node 3 Node 1

- hop count – counts the number of nodes that have seen this look up operation.
- route- starting with the name of the node that has started this look up then print the names of the nodes that have seen this look up operation.

Output:

1. The following configurations has to be run:
   a. node count = 10, m = 10
   b. node count = 100, m = 20

c. node count = 1000, m = 20

2. For each configuration, **the corresponding output files should be generated.** Each output file should contain the output for each key (as specified in the look up section).
3. For each configuration, calculate the average hop count and add it at the end of the corresponding output file. (eg:- average hop count = 5)

Summarized Requirements:

- Implement logics for
  - protocol/ChordProtocol.java

```
public void buildOverlayNetwork();
public void buildFingerTable();
public LookUpResponse lookUp(int keyIndex);
```

  - ChordProtocolSimulator.java
    - Look up all keys and find all the keys
- Run 3 configurations, generate output as described and calculate the average hop (save all the printing to files).
  - Node count=10, m=10
  - Node count=100, m=20
  - Node count=1000, m=20
- All the code must be well commented.

Notes:

- All the requirements must be met in your submission in order to book a Presentation Slot. If not, a FAIL grade will be given immediately.
- It is your choice to not attend the Group Session, all the question answered or information provided in the Group Session will not be answered again if you send me inquiries outside the Group Session (like email or devilry discussion). Only the questions and information that have not been provided in the Group Session are allowed.

References:

1. https://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf

Deliverables:

Via the Devilry system.

- A compressed file (zip) containing all your source code and documentation:
  - The source code must be well commented.
  - The documentation can be a simple help-me file explaining how to run your application, and explaining how the 3 parts mentioned in the assignments are implemented. Additionally, your documentation must explain how you distributed the workload among your group.
  - Output Results

- Ready to deploy jar file

Please Note:

- All submitted files will be checked for code plagiarism!

Submission Deadline: 23:59 on October 30, 2025