Chord

DNP Lab 5
Innopolis University, Fall 2022

Task

You have to implement a simplified version of the chord distributed hash table using grpc and protobuf.
This task can be done in a team of 2 people.

Additions are colored!

Requirements:

1. Implement a Node.
2. Implement a Client.
3. Implement a Registry
4. Use grpc to communicate between nodes, registry and the client.

Submission rules:

1. Your submission must contain:
    a. Your **Node.py** file.
    b. Your **Registry.py** file.
    c. Your **Client.py** file.
    d. Your **chord.proto** file.
2. Please stick to the given names.
3. Your submission may contain additional .py files if needed.
4. Submit the files as a single **.zip** archive (any name).
5. Only one person from the team (anyone) should submit your work.

Registry

Registry is responsible for registering and deregistering the nodes.
Important: as the registry uses random functions, make sure you initialized your random generator with seed 0. This will generate the same numbers every time you run the program. You must initialize your random generator with seed 0.

Registry features:

- It holds a dictionary of id and **address** port pairs of all registered nodes.
- Forms a finger-table for a specific node.
- Helps a node to join the ring.
- Every node knows about the registry.
- It is the only centralized part of the chord.
- There are no nodes in the ring at the start of the registry.

Run command:

python3 Registry.py <port> <m>

Registry has several command-line arguments:

- *port* - port number registry should run on.
- *m* - size of key (in bits). Thus, max size of the chord ring.

Run command example:
python3 Registry.py 5000 5

Functions:

- **register(ipaddr, port)**:
  This method is invoked by a new node to register itself. It is responsible to register the node with given **ipaddr** and **port**. I.e., assigns an **id** from identified space.
  Id is chosen randomly from range [0, 2^m-1]. If the generated id already exists, it generates another one until there is no collision.
  Returns:
  If successful: tuple of (node's assigned id, m)
  If unsuccessful: tuple of (-1, error message explaining why it was unsuccessful)
  It fails when the Chord is full, i.e. there are already 2^m nodes in the ring.

- **deregister(id):**
  This method is responsible for deregistering the node with the given **id**.
  Returns:
  If successful: tuple of (True, success message)
  If unsuccessful: tuple of (False, error message)
  It may fail, if there is no such registered id.

- **populate_finger_table(id):**
  This method is responsible for generating the dictionary of the pairs (id, ipaddr:port) that the node with the given **id** can directly communicate with.
  Also, this method should find the predecessor of the node with the given **id** - the next node reverse clockwise. For example, if there are no nodes between nodes with ids 31 and 2, then the predecessor of node 2 is node 31.
  Returns:
  Predecessor of the node with the given **id**, finger table of the node with the given **id** (list of pairs (id, ipaddr:port)).

- **get_chord_info():**
  This is the only method called by the client. This method returns the information about the chord ring (all registered nodes): list of (node id, ipaddr:port).
  Example:
  (2, 127.0.0.1:5002)
  (12, 127.0.0.1:5003)
  (20, 127.0.0.1:5004)

How to form a finger table:

Each node has a **finger table** containing $s \leq m$ entries. (s is the number of rows in a resulting finger table)
m - key size in bits (passed as a command-line argument).

FT_p - finger table of node with id *p.*

$FT_p[i]$ - i-th entry in finger table of node p.

1. Calculate $FT_p[i] = succ(p + 2^{i-1}) \bmod 2^m$ where $i \in [1, m]$
2. Remove duplicates.

Node

A node of a chord ring.

Node features:

- At the start, the node is not part of the chord.
- It first registers itself by calling the **register** function of the Registry.
- Requests the required ids from its successor.
  Example: Imagine a chord with nodes 5 and 10.
  IDs 6, 7, 8, 9 and 10 are stored on node 10.
  A new node appears with id 7. It is now responsible for ids 6 and 7. So, it must get keys and values (if any) from the node 10 and store them.

- Calls the **populate_finger_table** function of the Registry every second (only if it was successfully registered in the ring).
- Calls the **deregister** function of the Registry on exit.
- Stores some keys (data) of the chord.
- Able to save and remove data.

Run command:

python3 Node.py <registry-ipaddr>:<registry-port> <ipaddr>:<port>

Command-line arguments:
- *<registry-ipaddr>:<registry-port>* - ip address and a port number of the Registry.
- *<ipaddr>:<port>* - ip address and a port number this node will be listening to.

Functions:

- **get_finger_table()**
  This function is called by the client. It should just return the current finger table.

- **save(key, text):**
  Saves the key and the text on the corresponding node.
  - Calculates the hash of the given **key** and the id where the text should be stored as follows:

```
import zlib
hash_value = zlib.adler32(key.encode())
target_id = hash_value % 2 ** m
```

  - Finds the corresponding node following the **lookup procedure** (described below).

  Returns:
  If successful: (True, Node id it was saved on). Note that this id may differ from the calculated one.
  If unsuccessful: (False, Error message). For example, such a key may already exist.

- **remove(key)**
  Similar to the **save** method. But removes the key and the text from the corresponding node.
  Returns
  If successful: (True, Node id it was removed from)
  If unsuccessful: (False, Error message)

- **find(key)**
  Find the node, the key and the text should be saved on.
  Returns:
  If successful: (True, id, address:port) of the node this key is saved on.
  If unsuccessful: (False, error message). For example, if the given key was never

saved.

- **quit()**
  This method is responsible for calling the **deregister** method of the Registry to quit the chord and then shut down the Node.
  It is called when the Node receives the **KeyboardInterrupt** signal.
  Before shutting down:
  - It notifies its successor node (i.e., first node in its finger table) about the change of predecessor. In other words, it replaces successor's predecessor with its own predecessor
  - It transfers all data in its storage to its successor
  - It notifies its predecessor node about the change of successor, i.e., it replaces the predecessor's successor with its own successor.

Lookup procedure:

Let's suppose we asked node with id **p** to save key **k:**

- *If $k \in (pred(p), p]$* - current node (**p**) is the successor of a key **k**, thus **k** should be stored on the current node.
  - Suppose node 31 is a predecessor of node 2 in chord with m=5. If the calculated key **k** is 1, then this key should be saved in node 2

- Otherwise, to look up a key ***k***, node with id ***p*** will forward the request to node with index ***j*** from its finger table satisfying:

  $FT_p[j] \leq k < FT_p[j + 1]$

  - Suppose Node 24 with finger table $FT_{24}=[26, 31, 2, 16]$ is asked to save the key whose calculated id is 1. Then Node 24 selects the Node 31 since it is the farthest node that doesn't overstep calculated id (1).

Client

Client is a command line user interface to interact with the chord ring.
It continuously listens to the user's input and has several commands.

Run command:

python3 Client.py

Commands:

- **connect <ipaddr>:<port>**
  Establish a connection with a node or a registry on a given address.
  The client should be able to distinguish between a registry and a node.
  If there is no registry/node on this address, print the corresponding message.

If the connection is successful, print the corresponding message

- **get_info**
  Calls **get_chord_info()** if connected to a registry. Or **get_finger_table()** if connected to a node. And prints the result.

- **save “key” <text>**
  Example:
  save “some key” some “text” here
  key: some key
  text: some “text” here
  It tells the connected node to save the **text** with the given **key**. Prints the result.

- **remove key**
  It tells the connected node to remove the text with the given **key**. Prints the result.

- **find key**
  It tells the connected node to find a node with the **key**. Prints the result.

- **quit**
  Stop and exit client.

Example

1. Run Registry:

   ```
   python Registry.py 5000 5
   ```

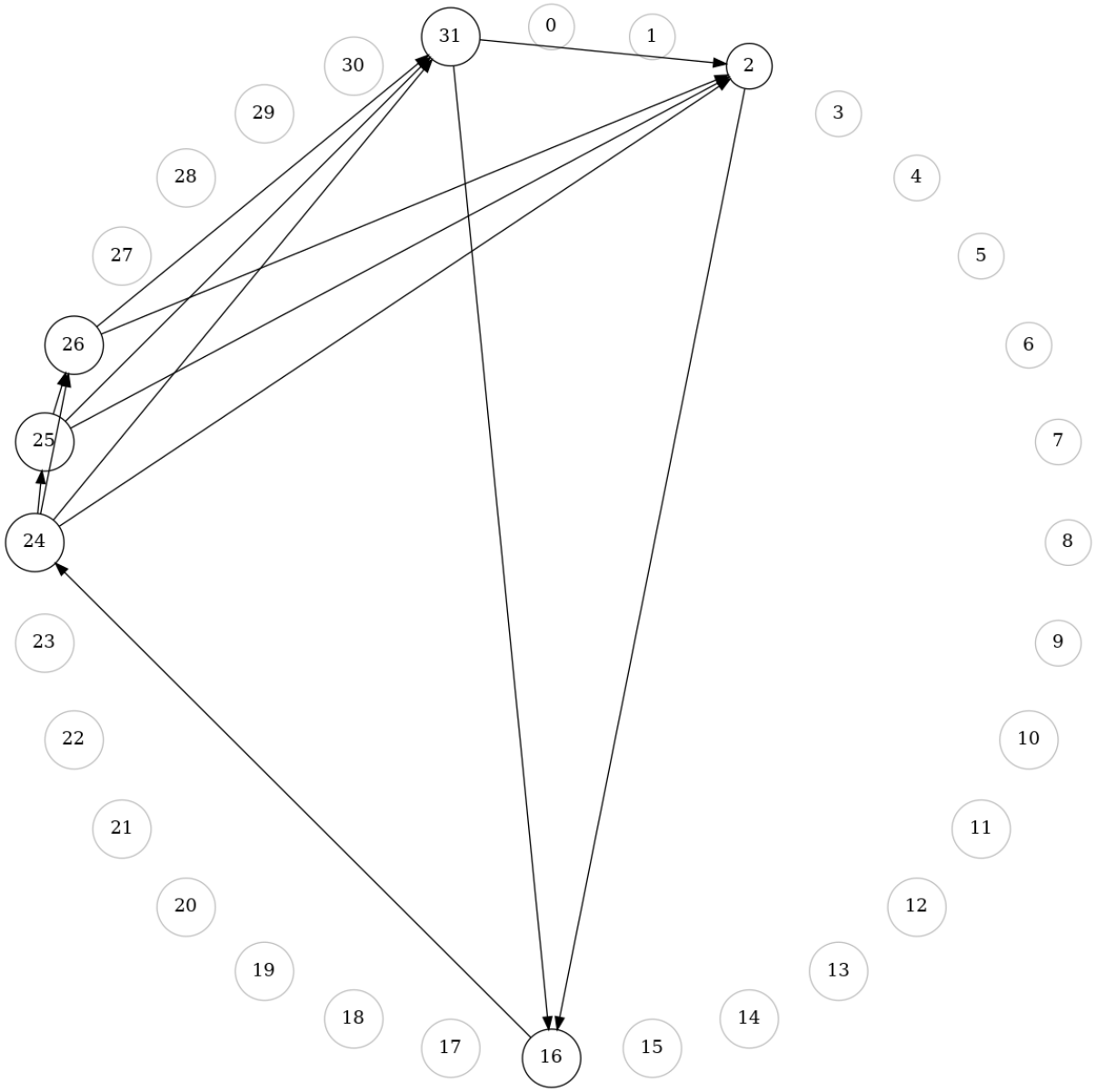
2. Run six nodes one by one:

   ```
   python3 Node.py 127.0.0.1:5000 5001
   python3 Node.py 127.0.0.1:5000 5002
   python3 Node.py 127.0.0.1:5000 5003
   python3 Node.py 127.0.0.1:5000 5004
   python3 Node.py 127.0.0.1:5000 5005
   python3 Node.py 127.0.0.1:5000 5006
   ```

Nodes are going to request the registry to give them an id. As they spawn one after another they connect to their neighbors as successors and predecessors:

```
assigned node_id=24, successor_id=24, predecessor_id=24
assigned node_id=26, successor_id=24, predecessor_id=24
assigned node_id=2, successor_id=24, predecessor_id=26
assigned node_id=16, successor_id=24, predecessor_id=2
assigned node_id=31, successor_id=2, predecessor_id=26
assigned node_id=25, successor_id=26, predecessor_id=24
```

If you set `random.seed(0)` on Registry and compute next id candidate as `random.randint(0,2**m_bits-1)` then likely you are going to have exactly the same ids allocated to your nodes. In the end your nodes gonna form following network, where arrows represent finger table relations:



Okay, now the network is up, we can try to store key-value data.

```
> connect 127.0.0.1:5000
Connected to Registry
> get_info
24:    127.0.0.1:5001
26:    127.0.0.1:5002
2:    127.0.0.1:5003
16:    127.0.0.1:5004
31:    127.0.0.1:5005
25:    127.0.0.1:5006
> connect 127.0.0.1:5001
Connected to Node
> get_info
Node id: 24
Finger table:
```

```
2:     127.0.0.1:5003
25:    127.0.0.1:5006
31:    127.0.0.1:5005
26:    127.0.0.1:5002
> save Kazan this-text-is-for-kazan
Saved on 16
> save Moscow moscow-text
Saved on 25
> save Minsk minsk-text
Saved on 2
> find Moscow
Key is located on 25
> remove Moscow
Key removed
> find Moscow
Key is localted on -1
> ^CTerminating
```