DNP Lab 07: Chord protocol (part 2)

Important

You need to have at least three files:

- **main.py**. Implements the user interface.
- **node.py.** Stores Node class which implements the behavior of p2p node of Chord
- **registry.py**. Stores Registry class which Implements the behavior of Registry node.

Introduction

In this assignment you will continue your last laboratory and implement some other important features of Chord protocol. Additions are highlighted.

1 Your main program

It should accept three arguments: *m*, *first_port,* and *last_port*.

- *m* represents the length of identifiers in chord (in bits) and has default value of 5.
- First, registry node is created whose constructor accepts *m* as argument
- *first_port* and *last_port* represent the range of port numbers that the nodes are going to use. If first_port=5001 and last_port=5011, then 11 nodes are created each of which listen at one of the 5001, 5002,…, 5011 ports.

Some commands of main program:

- *get_chord_info*. This command invokes the get_chord_info method of Registry class using RPC and returns data on current active nodes (i.e., dict)
- *get_finger_table p*. This command invokes get_finger_table method of the node instance (which is using port p) and finger table is returned (dict). Invocation should take place through RPC.
- *save p filename*. This command invokes the savefile(filename) method of the Node with port p, and the filename is saved on one of the node's store. Execution status and message should be printed out. For example:
  - ➢ (True, "filename is stored in Node 6")
  - ➢ (False, "filename already exists in Node 6")
- *get p filename*. This command invokes getfile(filename) method of the node with port p. The filename might have been saved in the given node or in one of the other nodes or maybe filename was never saved. The execution status and message should be printed out. For example:
  - ➢ (True, "Node 6 has filename")
  - ➢ (False, "Node 6 doesn't have filename")
- *quit p.* This command invokes quit method of the node instance (which is using port p). Invocation should take place through RPC. The execution status and message should be printed out. For example:
  - ➢ (True, "Node 6 with port 5000 was successfully removed")
  - ➢ (False, "Node 6 with port 5000 isn't part of the network")

2 Registry

It is responsible to register and deregister the nodes. Its behavior is implemented by Registry class which is a child of threading.Thread or multiprocessing.Process class.

- It holds dict of id and port pairs of all registered nodes.
- It has several methods:
  - **register(port)** method is registered as RPC service at predefined port and can be invoked by new node to register itself. It is responsible to register the new node with given port number, i.e., assigns id from identifier space. id is randomly chosen from [0, 2^m-1] range. Use random module and initialize it to seed=0 so your results will be the same as reference output! If newly generated id is the same as existing id, it again randomly selects another number until no collision
    - If successful, shall return tuple of (node's assigned id, optional message about the network size)
    - If unsuccessful, shall return tuple of (-1, error message explaining why it was unsuccessful)
      - It fails when for example when the Chord is full, i.e., there are already 2^m nodes given that m is the identifier length in bits
  - **deregister(id)** method is also registered as RPC service at predefined port and can be invoked by the registered node to deregister itself. It is responsible to deregister the node with given id. Returns tuple
    - (True, success message) upon successfully deregistering of the node with given id
    - (False, error message) upon failure. EX) no such id exists in dict of registered nodes
  - **get_chord_info()** method is also registered as RPC service and usually is invoked by your main program to get the information about chord: dict of node ids and port numbers.
  - **populate_finger_table(id)** method is responsible to generate the dict of the (id, port number) pairs that the requesting node can directly communicate with. It is registered as RPC service and can be invoked by the nodes.
    This method also returns the (id, port) of the predecessor node - the next node reverse clockwise. For example, if there are no nodes between nodes with ids 31 and 2, then predecessor of node 2 is node 31. So this method returns tuple in addition to dict of finger table.

3 Node

A usual p2p node of chord overlay. It is implemented by a Node class which is a child of either threading.Thread or multiprocessing.Process class. Node when created isn't part of chord. It first registers itself using RPC invoking **register(port)** method of Registry class. Then after a second it populates its finger table invoking the populate_finger_table method of Registry through RPC.

populate_finger_table also returns (id, port) of its predecessor node. Then following every second, Node updates its finger table and predecessor in the same way.

It has at least following method:

- **get_finger_table()** which returns the finger table of the node which is just a dict of node ids and port numbers. This method is registered as RPC service and accessible from the main program.

- **savefile(filename).** Calculates the hash of the given filename and then the id where this filename should be stored as follows:

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

  Now it uses lookup(target_id) procedure of chord protocol to fine where the filename should be saved. Following cases can exist
  - ➢ $target\_id \in (pred\_id, curr\_id]$

    The current node is the successor of target_id and thus the filename should be saved in current node. Ex)
    - Suppose node 31 is predecessor of node 2 in chord with m=5. If the calculated target_id for the filename is 1, then this filename should be saved in Node 2.
  - ➢ $target\_id \in (curr\_id, succ\_id]$

    The successor of current node is also successor for target_id and thus the file should be saved in current node's successor, i.e., in the first node in finger table. Then current node should call the successor's savefile method through RPC and pass the filename.
    - Suppose Node 24 with finger table FT_{24}=[26, 31, 2, 16] is asked to save the filename whose calculated target_id is 25. Then filename is saved in Node 26 who is the successor of Node 24.
  - ➢ Else, the current node should find such a node in its finger table, who is farthest from current node and doesn't overstep the target_id. Then selected node's savefile method is called through RPC and filename is passed. Ex)
    - Suppose Node 24 with finger table FT_{24}=[26, 31, 2, 16] is asked to save the filename whose calculated target_id is 1. Then Node 24 selects the Node 31 since it is the farthest node that doesn't overstep target_id.

  This method should return the
  - ➢ [True, "filename is saved in Node 2"]
  - ➢ [False, error_message]
    - Error message can be smth like: filename already exists in Node 2
- **quit()** which is responsible to call the deregister(id) method of Registry to quit the chord and then shut down the Node. quit() is also registered as RPC service and accessible from main program. 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 filenames in its storage to its successor
  - ➢ it notifies its predecessor node about the change of successor, i.e., it replaces predecessor's successor with its own successor.

  Returns the execution status and message
  - ➢ (True, success message) upon successfully deregistering of the node with given id
  - ➢ (False, error message) upon failure.

4 Example output

Important: Make sure that you initialized random module in registry.py with seed=0. **Note:** (*id*: *port*) combinations returned by **get_chord_info** can be different but make sure that only these ids: (26, 2, 24, 16, 31) are generated when you use seed=0. If you use

```
> python main.py 5 5000 5004
Registry and 5 nodes are created.

> get_chord_info
{24: 5000, 26:5001, 2:5002, 16: 5003, 31: 5004}

> get_finger_table 5000                    # node 24
{26:5001, 31: 5004, 2: 5002, 16: 5003}

# saving filename Kazan
> save 5000 Kazan
Kazan has identifier 22                    # this line is printed by main program
[True, ‘Kazan is saved in node 24’]        # this line is printed by main program

> save 5000 Moscow
Moscow has identifier 25
node 24 passed Moscow to node 26           # this line is printed by Node.py module
[True, ‘Moscow is saved in node 26’]

> get_finger_table 5002                    # node 2
{16: 5003, 24: 5000}

> get_finger_table 5003                    # node 16
{24: 5000, 2: 5002}

> save 5000 Rostov                         # node 24
Rostov has identifier 14
node 24 passed Rostov to node 2
node 2 passed Rostov to node 16
[True, ‘Rostov is saved in node 16’]       # this line is printed by main program

# getting filename Kazan
> get 5000 Rostov
Rostov has identifier 14
node 24 passed request to node 2           # this line is printed by Node.py module
node 2 passed request to node 16
[True, ‘node 16 has Rostov’]               # this line is printed by main program

> get 5002 Piter
Piter has identifier 5
node 2 passed request to node 16
[False, ‘node 16 doesn’t have Piter’]

> quit 5000
[True, ‘Node 24 with port 5000 was deregistered’]   # this line is printed by Registry.py module

> get_chord_info
{26:5001, 2:5002, 16: 5003, 31: 5004}        # chord has changed

> get_finger_table 5003
{26: 5001, 2: 5002}                          # node 16’s  table has changed

> get 5002 Kazan
node 2 passed the request to node 24         # this line is printed by Node.py module
node 24 passed the request to node 26        # this line is printed by Node.py module
[True, ‘Node 26 has Kazan’]                  # Kazan was migrated from 24 to 26
```