# Programming Assignment 5: Storing/Retrieving Objects from a DHT

## 1 Lab Task

In this lab, you will implement a DHT like CHORD and store and retrieve objects from it. All peers and objects are identified by IDs from 1 to 127. (You do not need to apply the hash function as in CHORD, you can use their IDs directly).

You can assume that there is a bootstrap server that maintains the ring as peers join, and can tell each peer where their place is in the ring. The bootstrap server is not doing any monitoring of the peers and the peers do not implement any heartbeat service.

Each peer will know the name of the bootstrap server from a command line argument. Each peer maintains the predecessor and the successor peer in the ring. Each peer also maintains a file with stored objects. You can assume that each peer will have a pre-populated object store. The file consists of lines looking like this: `clientID::objectID`

Each peer starts by contacting the *bootstrap server*. The bootstrap server maintains the ring, thus can tell the joining peer immediately where their place is in the ring. The bootstrap server will inform each of the peers from the ring that need to update their predecessor and the successor in the ring, i.e., the peer before and the peer after the insertion point.

**Example**: if the ring is 2 - 7 - 23 - 56 - 2, and peer 11 will be added to the ring, the bootstrap server will tell 11 that its predecessor is 7 and its successor is 23, will tell 7 that the successor is 11 and will tell 23 that its predecessor is 11.

A client interacts with the bootstrap server to store and retrieve objects in this ring. Mapping between objects and peers is done as in CHORD, an object with idObj between idPeer1 and idPeer2, will be stored on idPeer2.

You can assume that the maximum number of peers is 7. You should test your code with 7 peers and provide the testcases described below.

You can use `C/C++`, `Java`, or `Go`. Communication can be implemented with TCP. Communication should be reliable.

NOTE: You can assume that nobody fails and that searching by objects is done only on the ring (you will not implement the fingers table).

### 1.1 PART 1: Create a ring with 7 peers. (40 points)

After the bootstrap server started, each peer one by one will join the ring. The first peer that joins is always peer 1, but the remaining peers should be able to join in any order. For example: we want to be able to join peers 1, 5, 6, 8, 10, 23, 55, in this order, or 1, 10, 23, 55, 5, 6, 8.

For the testcases below, each peer will print on the screen the predecessor and the successor IDs as peers join the ring one by one. The bootstrap server will print the entire ring again, after each peer joined the ring. You can print the ring as in the example above.

**TESTCASE 1:** After the bootstrap server started, peers join the ring one by one, the list of peers joining the ring is with IDs in increasing order. The first peer is the peer with ID 1.

**TESTCASE 2:** Same as above, but the list of peers that will join the list is with IDs in random order.

### 1.2 PART 2: Storing and retrieving objects from the DHT (50 points)

A client program will store and retrieve objects from the DHT. The client communicates only with the bootstrap server. A client sends a REQUEST message that will contain: reqID, operationType, objectID, and clientID. reqID must be monotonically increasing per client, operationType is STORE or RETRIEVE, objectID and client ID are numbered from 1 to 127. When receiving the request to store an object, the bootstrap server forwards the request on the ring starting with peer 1, who then sends it to its successor and so on until it reaches the peer where the object must be stored. The peer where the object should be stored accepts the object, i.e., writes the ID of the received object to the Objects file and sends an OBJ_STORED message to the bootstrap server. The OBJ_STORED message should contain the object ID, client ID, and peer ID.

After storing a new object, the peer must print their Objects file which keeps track of all the objects they stored. For simplicity, we can assume that writing the object ID in the Objects file means storing the object. Mapping between objects and peers is done as in CHORD, if idObj is between idPeer1 and idPeer2, it will be stored on idPeer2.

**TESTCASE 3:** Client wants to store an object with a given ID. If the object is successfully stored, the client should print

```
STORED: <objectID>
```

**TESTCASE 4:** Client wants to retrieve an object that was previously stored with a given ID. If the object is successfully retrieved, the client should print

```
RETRIEVED: <objectID>
```

NOTE: An object should only be retrieved if its record matches both the clientID and the objectID.

**TESTCASE 5:** Client wants to retrieve an object with a given ID, but the object does not exist. As in TESTCASE 4 an object should only be retrieved if its record matches both the clientID and the objectID. If the object does not exist the bootstrap server should return -1 to the client, with the meaning that object was not found—this is detected if the request went around the ring without the object being found. The client should print

```
NOT FOUND: <objectID>
```

## 2 Implementation

You need to implement this algorithm in `C/C++`, `Java`, or `Go`, and your implementation must allow the user to configure the execution of the process. You will again be using Docker to package your program. Additionally, you must use Docker Compose as a container orchestrator. Instructions on how to use Docker Compose are provided in the Docker Tutorial. You will be provided with five Docker Compose files. They can be used as-is for each of the testcases. You must be able to run each of the Compose configurations and see the expected outputs printed to the screen. The expected interface for your containers is provided below. If you adhere to this interface, then Docker Compose will work seamlessly, without any changes. **Note:** You will find it easier to use the peer ID as the hostname for each of the peers.

```
Building:
    docker build . -f BootstrapDockerfile -t prj5-bootstrap
    docker build . -f PeerDockerfile -t prj5-peer
    docker build . -f ClientDockerfile -t prj5-client

      Running these in your project's directory should
      build your project's Bootstrap server, Peers, and Client images, respectively.
      They should copy over the relevant code to the image and compile
      the relevant component of your project.

Bootstrap Server Usage:
    docker run --name <hostname> --network <network> --hostname <hostname> prj5-bootstrap

Peer Usage:
    docker run --name <hostname> --network <network> --hostname <hostname> \
           prj5-peer -b server -o objects [-d delay]

Client Usage:
    docker run -it --name <hostname> --network <network> --hostname <hostname> \
           prj5-client -b server -t testcase [-d delay]

Arguments:
    --name <hostname>
      This specifies the name of the Docker container,
      this name should match the one in your hostsfile

    --network <network>
      This is the user-defined Docker network your project
      will run in, refer to tutorial in additional instructions
      to learn how to create one and why you need one.

    --hostname <hostname>
      This specifies the hostname of the Docker container,
      this name should match the one in your hostsfile. It can be used by other
      containers in the same network.

    -b server
      The hostname of the bootstrap server.

    -d delay
      The number of seconds to wait before joining after startup.

    -t testcase
      This argument only applies to client for testcases 3, 4, and 5.

    -o object
      Path to a file containing the object store of the peer.

Orchestration:
   docker compose -f [PATH TO COMPOSE FILE] up
```

# 3 Submission Instructions

Your submission must include the following files (10 points):

1. The **SOURCE** and **HEADER** files (no object files or binary)
2. A **MAKEFILE** to compile and to clean your project
3. A **README** file containing your name, instructions to run your code and anything you would like us to know about your program (like errors, special conditions, etc.)

Your submission should also include scripts, code, explanations on how to run the TESTCASES described above. To receive full credit for each part, you need to provide the required testcases. You must also be able to view the correct testcase output for each Docker Compose setup.

Submission is through gradescope.

# 4 Additional resources

You may find the following resources helpful

- Socket programming: `http://beej.us/guide/bgnet/`
- Unix programming links: `http://www.cse.buffalo.edu/~milun/unix.programming.html`
- C/C++ programming link: `http://www.cplusplus.com/`
- Docker tutorial: `https://github.com/iowaguy/docker-tutorial`