# Midterm Project-Chord, Part 1: DHT layer implementation

For the midterm project, you will be implementing the DHT layer of the Chord protocol in Part 1. You will be provided with C++ template code and will be expected to implement the RPCs and internal procedures used by Chord nodes.

You should first read the Chord Paper before beginning this assignment.

## Setup

To begin this assignment, you will need to install some libraries and packages on your machine and compile the code using the pthread library. We suggest running a docker virtual environment for this assignment.

Start by downloading and extracting the assignment data:

```
$ wget ntu.im/IM5057/chord-part-1.tar.gz
$ tar zxvf chord-part-1.tar.gz
```

The `chord-part-1` folder contains the following directories:

```
chord-part-1/
|
+-- chord/
    |
    +-- Makefile
    |
    +-- src/
    |
    +-- test_script.py
```

`src/` contains the template code for your implementation, and `test_script.py` is an example script for interacting with the Chord nodes via RPC calls.

To implement the RPCs, we will be using the rpclib library. To install rpclib, clone the github repository and build it:

```
$ git clone https://github.com/rpclib/rpclib.git
$ cd rpclib
$ mkdir build
$ cd build
$ cmake ..
$ cmake --build .
$ make install
```

After successfully installing rpclib, you can compile the template code using the `Makefile` under the `chord/` directory by running:

```
$ make
g++  -std=c++17 -Wall -pthread -o chord chord.cc -lrpc
```

To interact with the Chord nodes using a Python script, you will also need to install MessagePack RPC for Python:

```
$ pip install msgpack-rpc-python
```

You can then start a Chord node by running:

```
$ ./chord <ip> <port>
```

For example, to start two Chord nodes listening on port 5057 and port 5058 respectively, run:

```
$ ./chord 127.0.0.1 5057 & ./chord 127.0.0.1 5058 &
```

> Note that while a Chord node does not need to know its own IP address to run, its IP address (together with port) will be served as its "contact information" when interacting with other nodes. Other nodes may need to interact with it by calling the RPC using its IP address. In this assignment, we will use 127.0.0.1 for running all nodes as we will only be testing the Chord DHT locally. However, in Part 2 and Part 3, you will need to provide the IP address of the cloud instance you are running on when starting a Chord node, so that each Chord node running on an individual cloud instance will be able to reach each other.

After starting Chord nodes, you can interact with them by calling the RPCs using Python:

```
$ ./test_script.py
[b'127.0.0.1', 5057, 716540891]
[b'127.0.0.1', 5058, 1324994620]
```

We will discuss the details of these C++ and Python codes later on in the assignment.

# Chord implementation

## Implementing and Registering RPCs

The `rpcs.h` file contains the functions (including exposed RPCs and other internal procedures) of Chord. You should make your implemented Chord system correct by modifying only the `rpcs.h` file. Although you are encouraged to modify other files for experimentation, we will only use the `rpcs.h` file you submit for grading.

In our implementation, we represent the information of a Chord node using `struct Node`:

```
struct Node {
  std::string ip;
  uint32_t port;
  uint64_t id;
};
```

The `id` is a 32-bit identifier generated by hashing the `(ip, port)` pair.

All members of the `Node self` variable in `rpcs.h` are filled automatically when starting the Chord node. **You are not allowed to change the value of the `self` variable**. We can implement an RPC call for the Chord node which takes no parameters and returns its information:

```
Node get_info() { return self; }
```

After defining the new RPC call, we register it as an exposed RPC using the `add_rpc()` function:

```
  add_rpc("get_info", &get_info);
```

The two parameters it takes are the name and the function pointer of the RPC, and your RPC name should usually be the same as the function name.

## Inter-node RPC Call

A Chord node can invoke an RPC of another Chord node. For example, the `join()` function:

```
  rpc::client client(n.ip, n.port);
  successor = client.call("find_successor", self.id).as<Node>();
```

initializes an `rpc::client` instance with the IP and port of the Chord node to connect to, and then calls the `"find_successor"` RPC of the Chord node with parameter `self.id`, updating its successor with the return value. For more details on RPC calls, you can refer to the rpclib document.

## Handling Failed RPC Calls

If an RPC call fails due to an invalid IP or port, or because no Chord node is listening on the specified IP port, an exception will be thrown by rpclib. We should catch this exception to avoid terminating the entire process. For example, in the `check_predecessor()` function:

```
  try {
    rpc::client client(predecessor.ip, predecessor.port);
    Node n = client.call("get_info").as<Node>();
  } catch (std::exception &e) {
    predecessor.ip = "";
  }
```

if the RPC call fails, the exception will be caught and `predecessor.ip = ""` will be executed.

## Implementing and Registering Periodically Called Functions

Some functions in the Chord Protocol should be invoked periodically, such as the `check_predecessor()` function. We can register these functions to be periodically called using the `add_periodic()` function:

```
  add_periodic(check_predecessor);
```

Periodically called functions should have a void return type and take no parameters.

By default, these periodically called functions will be invoked every 2000 ms (2 s). If you want to specify this interval, you can provide the time interval (in ms) as the third parameter when starting the Chord node:

```
$ ./chord 127.0.0.1 5057 <interval>
```

# Tests

In addition to calling RPC between Chord nodes, you can also use the MessagePack RPC package in Python to call the RPC of Chord nodes. Some examples are provided in `test_script.py`.

In `test_script.py`, a helper function `new_client()` is provided for creating a new client instance with the IP and port of the Chord node to connect to.

```
client_1 = new_client("127.0.0.1", 5057)
client_2 = new_client("127.0.0.1", 5058)
```

Similar to the usage of rpclib in C++, after creating the client instance, you can use this client to call the RPC of the Chord node:

```
print(client_1.call("get_info"))
print(client_2.call("get_info"))
```

These two lines of code call the `"get_info"` RPC of two Chord nodes respectively and print the returned results.

`test_script.py` also provides an example for initializing the Chord ring:

```
client_1.call("create")
client_2.call("join", client_1.call("get_info"))
```

The first line calls the `"create"` RPC of Node 1 to create a Chord ring, and the second line calls the `"join"` RPC of Node 2, using the return value of the `"get_info"` RPC of Node 1 as a parameter, to allow Node 2 to join the Chord ring created by Node 1.

# Requirements

We will grade your implementation of Chord based on its functional completeness.

Firstly, you are only allowed to submit the `rpcs.h` file. Therefore, you must ensure that the `rpcs.h` file you submit can be successfully compiled with the original `chord.cc` and `chord.h` files to form a fully functional Chord.

Regarding RPC functions, the following requirements must be met:

1. Do not modify the `get_info()` function (and the RPC name it was registered to). This function will be used to test the correctness of your implementation.
2. You may modify the implementation of `"create"`, `"join"`, and `"find_successor"` RPCs, but you cannot change their parameter types and return types. These three RPCs will also be used to test the correctness of your implementation.
3. Apart from these, you are free to add any additional RPC functions and periodically called functions that you require.

Firstly, the test script will call the `"create"` RPC of any node to create the Chord ring, then call the `"join"` RPC of other nodes in any order to join the Chord ring:

```
client_0.call("create")

client_2.call("join", client_0.call("get_info"))
client_1.call("join", client_2.call("get_info"))
client_4.call("join", client_1.call("get_info"))
client_5.call("join", client_0.call("get_info"))
client_3.call("join", client_4.call("get_info"))
# ...
```

Next, the test script will pause for sufficient time for the periodically called functions of each node to maintain the successor, predecessor, finger table, and other information.

Next, the test script will call the `"find_successor"` RPC of any node and verify the correctness of the returned result:

```
ans = client_0.call("find_successor", 1234567)
# check ans
ans = client_1.call("find_successor", 1234567890)
# check ans
# ...
```

## 1. Correctness (40%)

Firstly, your implementation of Chord must always return the correct answer in the above testing method.

During this stage, we will test your Chord system using the `"get_info"`, `"create"`, `"join"`, and `"find_successor"` RPCs as mentioned above. We will set the interval for periodically called functions to 2000 ms. Moreover, our test script will pause for sufficient time between some RPCs. The specific rules are as follows:

| this RPC / next RPC | create | join | find_successor | get_info |
|---|---|---|---|---|
| **create** | X | 2 s | 2 s | 0 s |
| **join** | X | 2 s | 20 s | 0 s |
| **find_successor** | X | 2 s | 2 s | 0 s |
| **get_info** | 0 s | 0 s | 0 s | 0 s |

For example, we will wait for at least 2 seconds after calling a `"join"` RPC before calling the next `"join"` RPC. We will wait for at least 20 seconds after calling a `"join"` RPC before calling a `"find_successor"` RPC. We may call a `"get_info"` RPC immediately after calling a `"join"` RPC. We will not call a `"create"` RPC after calling a `"join"` RPC.

**Please note that if your Chord system fails to meet correctness, you will not receive credit for the subsequent parts.**

## 2. Message Complexity (30%)

Next, we will measure the message complexity of your implementation of Chord when processing `"find_successor"` requests. Your implementation of Chord must efficiently use the finger table to process `"find_successor"` requests in $O(\log n)$ message complexity. Specifically, in our testing, we will run at most 16 Chord nodes. Therefore, you can use a finger table of size 4 to record the information of the required nodes. **In a system with 16 Chord nodes, the average number of messages used by your implemented Chord system for each `"find_successor"` request must be less than 7.**

We will also measure the message complexity of your implemented Chord system when executing periodically called functions without handling any requests. **In a system with 16 Chord nodes, with no node joining or leaving for a long time and no external requests being handled, the total number of RPC calls per second sent in your Chord system must be less than 64.**

Please ensure that your Chord node only updates one entry in the finger table per periodically called function. If you update too many entries in the finger table in a short period of time, you may not meet the message complexity limit. Therefore, it is recommended to use a finger table of size 4. If you use a larger finger table, your Chord system may not be able to update the finger table within the time interval.

## 3. Fault Tolerance (15% + 15%)

Finally, your implemented Chord system must automatically recover and operate normally when a node fails. We have registered the `"kill"` RPC in the Chord code, and you can call this RPC in your test script to stop a Chord node:

```
client.call("kill")
```

Our test script will pause for enough time after calling this RPC to allow each node to complete stabilization. Then, we will continue to test the correctness of the `"find_successor"` RPC.

In the first stage, our test script will **kill at most one node every 20 seconds**. If your Chord system can operate correctly under these conditions, **you will receive a partial score of 15%**. The rules of time intervals between RPCs are as follows:

| this RPC / next RPC | create | join | find_successor | kill | get_info |
|---|---|---|---|---|---|
| **create** | X | 2 s | 2 s | X | 0 s |
| **join** | X | 2 s | 20 s | 20 s | 0 s |
| **find_successor** | X | 2 s | 2 s | 2 s | 0 s |
| **kill** | X | 40 s | 40 s | 20 s | 0 s |
| **get_info** | 0 s | 0 s | 0 s | 0 s | 0 s |

In the second stage, our test script will **kill at most two nodes every 20 seconds**. If your Chord system can operate correctly under these conditions, **you will receive the full score of 30%**. The rules of time intervals between RPCs are as follows:

| this RPC / next RPC | create | join | find_successor | kill | get_info |
|---|---|---|---|---|---|
| **create** | X | 2 s | 2 s | X | 0 s |
| **join** | X | 2 s | 20 s | 20 s | 0 s |
| **find_successor** | X | 2 s | 2 s | 2 s | 0 s |
| **kill** | X | 40 s | 40 s | 0 s / 20 s | 0 s |
| **get_info** | 0 s | 0 s | 0 s | 0 s | 0 s |

# Submission

Please put your `rpcs.h` in a folder named `<Student ID>_chord-part-1`, compress the folder as `<Student ID>_chord-part-1.zip`, and submit it to COOL.

```
<Student ID>_chord-part-1/
|
+-- rpcs.h
```

**Late submissions will not be accepted.**

# Notes

1. If a timeout error occurs during testing of Chord nodes, it may be due to either an implementation error or an overload of the machine caused by the number of Chord nodes. In this case, try reducing

the number of Chord nodes being tested or increasing the time interval for periodically called functions.

2. Please do not make the code for this assignment public on the internet. If you want to put it on GitHub, please set the repository to private.