# CS271 Winter 2023
# Final Project

Assigned: February 16, 2023
Due: Friday March 17

**PLEASE NOTE**:

- This programming assignment can be done in teams of TWO.

## 1 Introduction

In this final project, you will develop a distributed shared dictionary system replicated on a set of clients. The clients are responsible for maintaining a log of encrypted messages using **Raft**. Clients will be able to share multiple dictionaries with a subsets of other clients, and be able to put key-value pairs into a particular dictionary and get values from a particular dictionary via their corresponding keys.

### 1.1 Application Components

Since *Raft* is a *leader-based* consensus protocol, the project can be decomposed into three parts:

- Leader Election
- Normal Operations
- Fault Tolerance

In the first part, you will implement the leader election phase of the Raft protocol. In the second part, you will implement normal operations of the Raft protocol to maintain a log that will be used to create dictionaries and operate them. In the last part, you will deal with node failures and network partition failures within the Raft protocol while supporting normal operations.

You should prioritize the various parts of the assignment in the following order:

1. Implement Raft
2. Support persistent disk storage of Raft's state (e.g. in a file on disk)

3. Support site crashes and recovery from disk
4. Support replicated dictionaries with no privacy
5. Support replicated dictionaries with privacy using public-private key cryptography
6. Support network partitioning and recovery from a partition

## 1.2 Leader Election

There will be five clients starting as five followers at the beginning.

Following the Raft leader election protocol, there will be exactly one leader, and four followers after the leader election stage.

## 1.3 Normal Operations

In order to execute normal operations, you must correctly implement the Raft protocol first. For the privacy part of this project, you will need to generate public-private key pairs, and encrypt and decrypt data, which you are free to use libraries like `RSA` to perform key generations and encryptions.

### 1.3.1 Clients

Clients are interested in maintaining copies of some dictionaries and executing put and get operations on copies of specific dictionaries. Each of the five clients will maintain and persist the following state on disk:

- The client's personal public and private keys
- The public keys of all other clients (you can either choose to publish this to the log at startup or keep it separately)
- A copy of the replicated log and other states needed for Raft (e.g. `currentTerm` & `votedFor`)

At initialization, the clients will all be directly connected to each other in a mesh network. Note that due to failures, this initial configuration may change (more below).

### 1.3.2 Creating a dictionary

- The user can decide at any time to have a client create a new dictionary that is replicated among a subset of clients. Any client within this subset is considered a member of the shared dictionary and has access to its contents

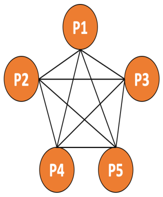

Figure 1: All processes directly connected to each other at initialization

- To create a new dictionary, the client will first create a globally unique `dictionary_id` for that dictionary (e.g. using its pid + a persistent counter)
- Then, the client should generate a public-private key pair for that dictionary. It will create a version of the dictionary's private key for each member by encrypting the dictionary's private key with that member's public key
- To distribute the info regarding this new dictionary, the client will add the `dictionary_id`, the client IDs of the members of the dictionary, the dictionary's public key, and all versions of the encrypted dictionary private key to the log
- A client will create the new dictionary (i.e. the client's Raft state machine processes the log entry) once the new dictionary's log entry is considered committed by Raft

#### 1.3.3 Operating on a dictionary

- At any point in time, a user should be able to have a client issue operations on a particular dictionary that the client has access to / is a member of. There are 2 types of operations:
  - **Putting key-value pairs into the dictionary**
    Put a key-value pair (e.g. put(amr, cs271), where "amr" is the key and "cs271" is the value) into a particular dictionary
  - **Getting a value from the dictionary via a key**
    Get a value from a particular dictionary with the value's corresponding key (e.g. get(amr) to get the value "cs271") and print the value to console
- The client should encrypt an operation with the public key of the dictionary that the operation is for
- It should then add the encrypted operation, the `dictionary_id`, and its own `client_id` to the log

- Any client who is replicating the log and is a member of the dictionary with `dictionary_id` should decrypt the operation and apply the operation to its copy of the specified replicated dictionary once the operation's log entry is considered committed by Raft
- While a client that is not a member of the dictionary can technically still encrypt operations with the dictionary's public key and add them to the log, members of the dictionary shouldn't perform a non-member's operation on their replicated dictionaries

### 1.3.4 Replicated Log

You will create a replicated log that will be maintained on all five clients **via the Raft protocol**.
We do not necessarily trust all the clients, thus to detect any potential tampering with the content of entries, we require that each entry contain **a hash of the previous entry** in the log. When a client is replicating new entries in the log, it should **verify** that the hash of every new entry matches the hash of the content for that entry's previous entry in the log. If the hashes don't match, the client should reject that entry instead of replicating it to the client's log. You are free to use hashing libraries like `SHA256` to create the hashes.

The replicated log will contain the following types of entries:

- **Create dictionary.** Containing the `dictionary_id`, dictionary member `client_id`s, dictionary public key, encrypted versions of the dictionary private key with the members' public keys
- **Put operation.** Containing the `dictionary_id`, the issuing client's `client_id`, and the key-value pair (to be put into the dictionary) encrypted with the dictionary's public key
- **Get operation.** Containing the `dictionary_id`, the issuing client's `client_id`, and the key (to get a corresponding value from the dictionary) encrypted with the dictionary's public key

You do not necessarily have to follow the structure above and you may split up the data according to what best suites your application. Note that in addition to application data, each entry will also need to contain data required by Raft (e.g. the `term` and `index`).

## 1.4 Fault Tolerance

The clients should be able to perform normal operations even if the following failures happen:

### 1.4.1 Node Failure

- **Follower failure.** This is where a non-leader node fails. After a follower fails, the program should still be able to perform normal operations that don't require that follower.
- **Leader failure.** This is where a leader fails. After leader failure, Raft will begin a new term with leader election after which it should be able to perform normal operations.

### 1.4.2 Network Partition

Due to network failures, the clients can be decomposed into multiple partitions. In one scenario, one of the partitions has a majority of clients (i.e. $\geq 3$ clients). If the leader is amongst them then this scenario behaves like a follower failure and if it is not amongst them then this scenario behaves like a leader failure and a new leader must be elected to continue normal operation. In another scenario, none of the partitions have a majority. In this case, no partition can elect a leader and thus normal operation stops until these partitions join to form a majority.

### 1.4.3 Node Recovery

After a node recovers from a failure or connects to the majority partition after a network partition, it needs to update its log and resume normal operation. You can decide how to handle this recovery but we expect you to store the log as well as any private and public keys on disk. This will allow you to recover from a reasonable state in case all processes fail. The client that just recovered needs to be able to read any messages it may have missed. This should be the case even if the client was added to a group while it was in a crashed state.

## 2 Implementation Details

- Please add a delay 3 second delay before receiving each message to simulate network delays.
- You can use an existing libraries in your language of choice to do public-private key cryptography

## 3 User Interface

The user must be able to input the following commands:

1. `create [<client_id>...]` : A client receiving this command should create a new dictionary with the clients from the list of `client_id`s specified in the command as its members. The client should generate a `dictionary_id` for the dictionary.

2. `put <dictionary_id> <key> <value>` : A client receiving this command should put the pair `key`:`value` into the dictionary with `dictionary_id`.

3. `get <dictionary_id> <key>` : A client receiving this command should get the value corresponding to `key` from the dictionary with `dictionary_id` and print the value to console.

4. `printDict <dictionary_id>`: This command should print the `client_id`s for all of the dictionary members as well as the content of the dictionary with `dictionary_id`.

5. `printAll`: This command should print the `dictionary_id`s for all dictionaries that the client is a member of

6. `failLink <dest>` : This command must emulate a communication link failure between a source process and a `dest` process. For example: if we want the communication link between P1 and P2 to fail, we will use this interface as a user input on process P1 by running `failLink P2`. Your program should then treat the link between the two nodes P1 and P2 as failed, and hence, should not send any message between the two nodes. For simplicity, links are considered bidirectional and breaking a link allows neither the source process nor the `dest` to communicate with each other.

7. `fixLink <dest>`: This is a counterpart of failLink input. This input fixes a broken communication link between two processes, upon which the two nodes will be able to communicate with each other again. This input will be provided on the source process.

   A suggestion to simulate such a behaviour could be to use a dictionary data structure that has the network links in the system and has an active or non-active boolean. You can use such a map to check if the network link is active before sending out any message. This is one suggestion; you can use any other technique as long as the behaviour of network failure and fixing of it is emulated.

8. `failProcess`: This input kills the process to which the input is provided. You will be asked to restart the process after it has failed. The process should resume from where the failure had happened and the way to do this would be to store the state of a process on disk and reading from it when the process starts back.

Generally, you should print any information that reflects changes in the state of the system/clients to the console. This includes but is not limited to:

- Any messages sent and received as part of the Raft protocol
- When the Raft state machine processes a log entry
- Any changes to the replicated log

- Any send or receive failures due to network partitions or failed processes
- Outcome of verifying the hash of a new entry
- Confirmation that the program registered a user input

## 4 Deadlines and Deployment

**This project will be due on Friday March 17**. Signup sheet for demo time slots and zoom link will be posted on piazza. Gradescope submission is due at midnight on the same day as demos. For demoing this project over Zoom, you can deploy your code on several machines. It is also acceptable if you just use several processes in the same machine to simulate the distributed environment.