Big Data Processing Technologies Project 2: Distributed Lock Design Report

Fuming Zhang, 118033910025

1 Overview

In this project, we are supposed to design and implement a simple consensus system, which satisfies the following requirements:

1) Contain one leader server and multiple follower server.
2) Each follower server has a replicated map, and the map is consisted with the leader server.
3) The key of the map is the name of the distributed lock, and the value is the client ID who owns the distributed lock.

Basically, each server corresponds to a client and each client could perform any of the following three operations:

1) For preempting a distributed lock, if the lock doesn't exist, preempt success. Otherwise, preempt fail.
2) For releasing a distributed lock, if the client owns the lock, release success. Otherwise, release fail.
3) For checking a distributed lock, any client can check the owner of a distributed lock.

Due to the different roles of the server, when processing client requests, the follower servers should send all preempt or release requests to the leader. While dealing with checking operations, the follower server could access its map directly and sends the results to the clients. When it comes to the leader server, it should modify its map and sends a update request to all follower servers if needed when it handles preempt or release request. Thus when a follower server receives a request propose, it should modify its local map, respond to the leader server and return results to pending clients. What's more, when a client is initialized, we should define the IP address of the target server and generate the client ID information based on the user information(UUID).

The above is the description of this project. Following these instructions and the reference data structure provided, we designed and implemented a simulated distributed lock based on threading and socket modules in Python 3. In our design and implementation, we provide two types of locks, mutex and readers-writer lock. In addition, we have also provided two levels of consistency guarantee. The specific design and implementation will be discussed in detail in Section **??** and **??**, respectively.

- *Project link: https://github.com/AlexChang/DistributedLock*
- *E-mail: zhangfuming-alex@sjtu.edu.cn*

2 Deisgn

In this section, we are going to introduce the design of our consensus system. According to the project requirements, we should first design the following three classes, client, leader and follower.

In class client, its initialization should include recording its assigned IP address and generating its own user information (UUID). In addition to the above initialization, it also needs to provide an interface for the main process to specify its corresponding server. It will then communicate with the server via the IP address and port of the server specified here. What's more, it also needs to provide the interfaces to perform preempt lock, release lock and check lock these three kinds of operations, each of which accepts only one parameter, the lock key.

In class leader, its initialization should include recording its assigned IP address and port, generating its own server information (UUID), initializing lock map and follower server address list and initializing the lock according to the specified lock type. Next, it needs to provide an interface to start its listening service. When the service receives the request from the specified port, it will hand over the subsequent processing steps to a new thread for further processing. Considering that there exist two different types of requests, we need to design separate processing interfaces for them, one is to process the request sent by the client, and the other is to process the request sent by other servers (i.e., follower servers).

More specifically, when processing a client request, according to different request instructions, further processing is performed by preempt lock, release lock, and check lock these three specific operation interfaces. Here we need to provide an additional interface for sending update requests to all the follower servers. When dealing with server-oriented requests, it also hands over the subsequent processing steps to specific operation interfaces according to request instructions.

The class follower and class leader are partially similar. It differs from class follower initialization in that it records the IP and port of the leader server instead of the follower servers. It also needs to provide an interface to start its listening service. The interface that handles user requests is similar to class leader, but the interface to deal with server-oriented requests is slightly different. It needs to provide an interface to handle update requests.

In addition to the three classes mentioned above, we also need the main function, in which we instantiate the clients and servers, configure their ports, start and add the corresponding server to the client, and then randomly generate the client's requests for simulation. Finally, we shut down all started servers.

Finally, we have to design a readers-writer lock. The threading library in Python only provides mutex lock, spin lock and conditional lock and based on mutex and conditional lock we could implement our own readers-writer lock. The basic idea of the readers-writer lock is that only one writer acquires the write lock or a number of readers acquire the read lock at a certain moment. The class rwlock should mainly provide these four following interfaces, write_acquire, write_release, read_acquire and read_release.

3 Implementation

In this section, we will discuss the implementation of our distributed lock in detail.

As for class client, the execution of the three operations of preempting lock, releasing lock and checking lock are quite similar. It first stores the request information in a dictionary which contains three keys: type, op and args. Type indicates the request type and its possible value is 'client' or 'server'. Op indicates the specific instruction and its possible value is 'try_lock', 'try_unlock' or 'own_the_lock'. Args is a dictionary containing two key-value pairs. One of them is lock_key and the other one is client_id. Next, we convert the dictionary containing the request information into json format and then encode it with utf-8. We send the above request through the connection established with the server listening port. After receiving the result, we decode it and get the boolean type return value.

In class leader, whenever it receives a request from its listening port, it will open a new thread for subsequent processing. In the specific handler function handle_request, it checks the type field of the request. If its value is 'client', the subsequent processing will be accomplished by handle_client_request. If it is 'server', it is handed over to handle_server_request. In client request handler function handle_client_request, it tries to acquire a mutex or writer lock on the lock map before executing the preemt_lock and release_lock operations. As for which type of lock will be acquired, it depends on the parameter specified during server initialization. Moreover, before executing check_lock operation, it will try to acquire nothing or the reader lock. After the operations, the applied lock will be released. In server request handler function handle_server_request, it also tries to acquire the lock before executing the preemt_lock and release_lock operations and releases the lock after these operations.

As for the specific check_lock operation, it will check whether the lock map contains the key whose value equals to lock_key. If so, it will judge whether its corresponding value is client_id. The result will be true if it is true, and the result will be false otherwise. As for the preemt_lock operation, if there is no entry with the key whose value equals to lock_key, we update the lock map and request all follower server to update the lock map. As for release_lock, it first calls the check_lock operation to check whether client_id owns the lock with lock_key or not. If so, we remove this item from the lock map and request all follower server to update the lock map.

In the above two operations, we need to request all the follower servers to update the lock map. Thus in function request_to_update, we set the value of type to 'server', op to 'request', and args to the lock map to build the request. When we actually process the update request, we will open a new thread for each request and there are two ways to do this. One is to wait for the update requests to complete before returning the result of this operation, which will reduce certain concurrent processing capability, but guarantees strong consistency; the other is to return the operation result immediately, which improves the concurrent processing ability but leads to weak consistency.

In class client, the handler function handle_request is the same as the function in class leader. The client request handler function handle_client_request is also almost the same. The difference is that it does not try to acquire a mutex or writer lock on the lock map before executing the preemt_lock and release_lock operations. The attempt to acquire the lock is placed in server request handler function handle_server_request. This approach reduces consistency but avoids deadlocks. In the above function, it handles the update request from the leader server.

The specific check_lock operation is exactly the same as the function in class leader. As for the preemt_lock and release_lock operations, it forwards the request sent by the client to the leader server and returns the returned result to the client. As for the handle_update_request operation, it updates the local lock map based on the given lock map parameter.

In our main process function, we randomly generate the client's requests for simulation. These generated requests can be sent simultaneously or at regular intervals. Based on our test, when there are 1 leader server and 3 follower servers, if the try_lock or release_lock request sent by the client succeeds, it takes 0.012 seconds to complete the update of all server lock maps under strong consistency constraints.

Finally, we will briefly introduce the readers-writer lock we implemented. We use the variable *state* to record the current lock state, 0 means no lock acquired, a negative number means write lock acquired and a positive number means read lock(s) acquired. Since only one write lock can be obtained at a certain moment, the minimum value of *state* is -1. When applying for a lock, if the lock cannot be acquired immediately, we block it with the corresponding conditional lock and add 1 to the corresponding type of wait count variable. If a lock is released, we will notify the corresponding conditional lock and release the previous blockage. It is worth mentioning that our readers-writer lock is designed to be write-first by default.

4 Conclusion

In this project, we have designed and implemented a simple consensus system. Based on threading and socket modules in Python 3, we provide two types of locks, mutex and readers-writer lock, where the latter one could provide better concurrent control in such a simulated distributed lock. In addition, we have also provided two levels of consistency guarantee. After our tests, by analyzing the log information output by the program, our distributed lock can work as expected, so we think we have completed the project requirements.

ACKNOWLEDGMENTS

The author would like to thank Prof. Chentao Wu and Bin Yao and TA Pengcheng Zhang and Junqing Gu for their instructions on this work.