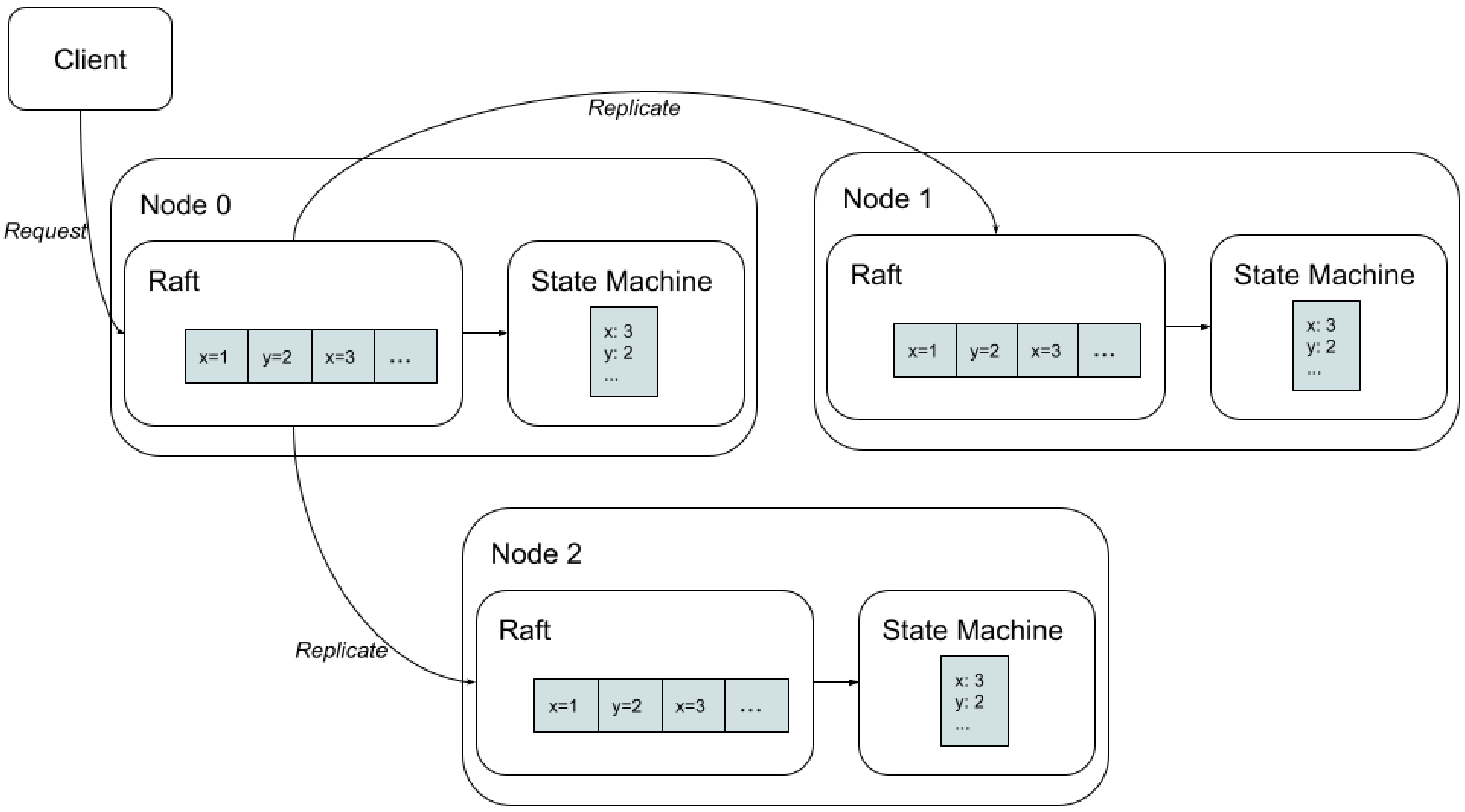
Lab 3: Fault-tolerant Key-Value Store with Raft

Due: 11-29-2021 23:59 (UTC+8)

Introduction

In this lab, you will design a Raft library to implement a fault-tolerant distributed key-value storage system. The library decouples the consensus algorithm from the replicated state machine. The user of the library only needs to care about the implementation of a single node state machine, e.g. a single node key-value store. With this library, we can easily extend a single machine system to a distributed system with fault tolerance.



A replicated system achieves fault tolerance by replicating the state on multiple machines. For example, a replicated key-value store copies all the key-value pairs to multiple machines. The replication mechanism makes the system available even if some of the servers crash (or encounter a network failure). However, the key challenge for implementing a replicated system is to keep the replicas consistent, which can be solved by consensus algorithms such as Paxos or Raft.

Raft is a consensus algorithm that is famous for its understandability. It's equivalent to Paxos in fault tolerance for replicated systems. Raft decomposes the consensus problem into relatively independent subproblems, which are much easier to understand. The key data structure in Raft is the **log**, which organizes the clients' requests into a sequence. Raft guarantees all the servers will **apply the same log commands in the same order**, which means the servers will all be in a consistent state. If a server fails but later recovers, Raft takes care of bringing its log up to date. And Raft can work as long as at least a majority of the servers are alive and connected.

Raft implements consensus by first electing a leader among the servers (part 1), then giving the leader authority and responsibility for managing the log. The leader accepts log entries from clients, replicates them on other servers, and tells servers when it is safe to apply log entries to their state machines (part 2). The logs should be persisted on the non-volatile storage to tolerate machine crashes (part 3). And as the log grows longer, Raft will compact the log via snapshotting (part 4).

This lab will follow the description of the Raft paper. **So, make sure you have read and understood the Raft paper of the extended version (especially section 5 and section 7) before coding.** And you will find that **Figure 2 and Figure 13 in the raft paper can cover most of your design in this lab**.

There are 5 parts in this lab.

- In part 1(20 points), you will implement the leader election and heartbeat mechanism of Raft.
- In part 2(45 points), you will implement the log replication protocol of Raft.
- In part 3(20 points), you will persist Raft log.
- In part 4(5 points), you will implement the snapshot mechanism of Raft.
- In part 5(10 points), you will implement a distributed key-value store based on your Raft library.

Each part relies on the implementation of the prior one. So you must implement these parts one by one.

If you have any questions about this lab, please feel free to let the TA know: Mingcong Han (mingconghan@sjtu.edu.cn).

IMPORTANT: You may take more than 12 hours to complete this lab. Start as early as possible!

Hope you can enjoy the lab!

Getting started

Please backup all of your prior labs' solutions before starting this lab.

```
% cd cse-lab
% git commit -a -m "upload lab2"
```

Then, pull this lab from the repo:

```
% git pull
```

Next, switch to the lab3 branch:

```
% git checkout lab3
```

Merge with lab2, and solve the conflicts.

```
% git merge lab2
```

After merging the conflicts, you should be able to compile the new project successfully:

```
% chmod -R o+w `pwd`
% sudo docker run -it --rm --privileged --cap-add=ALL -v `pwd`:/home/stu/cse-lab shenjiahuan/cselab_env:1.0
/bin/bash
% cd cse-lab
% make clean && make
```

Overview of the code

You will mainly modify and complete the codes in `raft.h`(part 1 - part 4), `raft_protocol.h, raft_protocol.cc` (part 1 - part 4), `raft_state_machine.h, raft_state_machine.cc` (part 5) and `raft_storage.h`(part 3 - part 4).

There are 4 important C++ classes you need to pay attention to.

The first two classes (`raft_command` and `raft_state_machine` in `raft_state_machine.h`) are related to the state machine. We have already implemented these two classes for testing your Raft implementation in the first four parts. And you will implement your own in part 5. But you still need to check the interfaces provided by them in the early parts. For example, you will use the `raft_state_machine::apply_log` interface to apply a committed Raft log to the state machine.

```
class raft_command {
public:
    virtual ~raft_command();

    // These interfaces will be used to persistent the command.
    virtual int size() const = 0;
    virtual void serialize(char* buf, int size) const = 0;
    virtual void deserialize(const char* buf, int size) = 0;
};

class raft_state_machine {
public:
    virtual ~raft_state_machine();

    // Apply a log to the state machine.
    virtual void apply_log(const raft_command &cmd) = 0;

    // Generate a snapshot of the current state.
    virtual std::vector<char> snapshot() = 0;
    // Apply the snapshot to the state mahine.
    virtual void apply_snapshot(const std::vector<char>&) = 0;
};
```

And the `raft` class (in `raft.h`) is the core of your implementation, which represents a Raft node (or Raft server). `raft` is a class template with two template parameters, `state_machine` and `command`. Remember we're implementing a raft library that decouples the consensus algorithm from the replicated state machine. Therefore, the user can implement their own state machine (e.g. a kv store) and pass it to the Raft library via the two template parameters.

The user ensures the `state_machine` inherits from `raft_state_machine` and the `command` inherits from `raft_command`. So you can use the interfaces provided by the two base classes in your implementation.

```
template<typename state_machine, typename command>
class raft {
public:
    raft(
        rpcs* rpc_server,
        std::vector<rpcc*> rpc_clients,
        int idx,
        raft_storage<command> *storage,
        state_machine *state
    );
    ~raft();

    // start the raft node.
    // Please make sure all of the rpc request handlers have been registered before this method.
    void start();

    // stop the raft node.
    // Please make sure all of the background threads are joined in this method.
    // Notice: you should check whether is server should be stopped by calling is_stopped().
    //         Once it returns true, you should break all of your long-running loops in the background
threads.
    void stop();
    // send a new command to the raft nodes.
    // This method returns true if this raft node is the leader that successfully appends the log.
    // If this node is not the leader, returns false.
    bool new_command(command cmd, int &term, int &index);

    // returns whether this node is the leader.
    bool is_leader(int &term);

    // save a snapshot of all the applied log.
    bool save_snapshot();
}
```

And the last important class is `raft_storage`, which you will complete to persist the Raft log and metadata. `raft_storage` is also a class template with a template parameter named `command`, which is the same as the template parameter of `raft` class. And you can use the interface provided by `raft_command`, such as `size`, `deserialize` and `serialize` to implement the log persistency.

```
template<typename command>
class raft_storage {
public:
    raft_storage(const std::string &file_dir);
}
```

Notice: You must not change the constructor definition of these classes.

Understand the `raft` class

Now, let's first walk through how `raft` works.

Our `raft` algorithm is implemented **asynchronously**, which means the events (e.g. leader election or log replication) should all happen in the background.
For example, when the user calls `raft::new_command` to append a new command to the leader's log, the leader should return the `new_command` function immediately.
And the log should be replicated to the follower asynchronously in another background thread.

A `raft` node starts after calling `raft::start()`, and it will create 4 background threads.

```
template<typename state_machine, typename command>
void raft<state_machine, command>::start() {
    RAFT_LOG("start");
    this->background_election = new std::thread(&raft::run_background_election, this);
    this->background_ping = new std::thread(&raft::run_background_ping, this);
    this->background_commit = new std::thread(&raft::run_background_commit, this);
    this->background_apply = new std::thread(&raft::run_background_apply, this);
    ...
}
```

The background threads will periodically do something in the background (e.g. send heartbeats in `run_background_ping`, or start an election in `run_background_election`).
And you will implement the body of these background threads.

Besides the events, the RPCs also should be sent and handled asynchronously. If you have tried
the RPC library in lab2, you may know that the RPC call provided by the lab is a synchronous version, which means the caller thread will be blocked until the RPC completes. To implement an asynchronous RPC call, this lab also provides a thread pool to handle asynchronous events. For example:

```
thread_pool->addObjJob(this, &raft::your_method, arg1, arg2);
```

To test your implementation, you can type:

```
% ./raft_test partX
```

And you can change the `partX` to the part you want to test, e.g. `part1` or `part2`.

Part 1 - Leader Election

In this part, you will implement the **leader election** protocol and **heartbeat** mechanism of the Raft consensus algorithm. And you can refer to Figure 2 in the raft paper to implement this part.

You'd better follow the steps:

1. Complete the `request_vote_args` and `request_vote_reply` class in `raft_protocol.h`. Also, remember to complete the marshall and unmarshal function in `raft_protocol.cc` for RPCs.
2. Complete the method `raft::request_vote` following Figure 2 in the Raft paper (you may also need to define some variables for the `raft` class, such as commit_idx).
3. Complete the method `raft::handle_request_vote_reply`, which should handle the RPC reply.
4. Complete the method `raft::run_background_election`, which should turn to candidate and start an election after a leader timeout by sending request_vote RPCs asynchronously.
5. Now, the raft nodes should be able to elect a leader automatically. But to keep its leadership, the leader should send heartbeat (i.e. an empty AppendEntries RPC) to the followers periodically. You can implement the heartbeat by implementing the AppendEntries RPC (e.g. complete `append_entries_args`, `append_entries_reply`, `raft::append_entries`, `raft::handle_append_entries_reply`, `raft::run_background_ping`).

You should pass the 2 test cases of this part. (10 points + 10 points)

```
% ./raft_test part1
Running 2 Tests ...
Test (part1.leader_election): Initial election
Pass (part1.leader_election). wall-time: 5.15s, user-time: 0.01s, sys-time: 0.03s
Test (part1.re_election): Election after network failure
Pass (part1.re_election). wall-time: 4.58s, user-time: 0.03s, sys-time: 0.05s
Pass 2/2 tests. wall-time: 9.73s, user-time: 0.04s, sys-time: 0.08s
```

Hints:

- You can run a single test case by its name, for example, `./raft_test part1 leader_election` will only check the `part1.leader_election` test case.
- We provide a macro in `raft.h` named `RAFT_LOG`, you can use this macro to print the system log for debugging. The usage of this macro is the same as `printf`, e.g. `RAFT_LOG("Three is %d", 3);`. But it will provide additional information, such as node_id and term, in the console.
- Be careful about the timeouts. We suggest use 150ms for heartbeat (i.e. send heartbeat every 150ms), 300 - 500ms for follower election timeout (i.e. start an election if no RPC recieved in 300ms - 500ms), and 1s for candidate election timeout (i.e. restart an election if no majority votes for it in 1s).
- You can send asynchronous RPC via the thread_pool. For example, to send an request_vote RPC, you can use : `thread_pool->addObjJob(this, &raft::send_request_vote, target, args);`
- Use the big lock (e.g. use `std::unique_lock<std::mutex> lock(mtx);` at the beginning of all the events) to avoid concurrent bugs.
- The background threads should sleep some time after each loop iteration, instead of busy-waiting the event.
- You don't have to implement all of the rules in AppendEntries RPC in this part (e.g. no need for a log). You only need to implement the heartbeat.
- You don't have to worry about the persistency issue until part 3.
- If your program crashes, try debugging the dumped core using GDB.

Part 2 - Log Replication

In this part, you will implement the `log replication` protocol of the Raft consensus algorithm. Still, you can refer to Figure 2 in the raft paper.

Recommended steps:

1. Complete `raft::new_command` to append new command to the leader's log.
2. Complete the methods related to the AppendEntries RPC (e.g. `raft::append_entries`, `raft::handle_append_entries_reply`).
3. Complete `raft::run_background_commit` to send logs to the followers asynchronously.
4. Complete `raft::run_background_apply` to apply the committed logs to the state machine.

You should pass the 7 test cases of this part. (10 points * 2 + 5 points * 5)

```
% ./raft_test part2
Running 7 Tests ...
Test (part2.basic_agree): Basic Agreement
Pass (part2.basic_agree). wall-time: 1.07s, user-time: 0s, sys-time: 0.03s
Test (part2.fail_agree): Fail Agreement
Pass (part2.fail_agree). wall-time: 3.81s, user-time: 0.01s, sys-time: 0.07s
Test (part2.fail_no_agree): Fail No Agreement
Pass (part2.fail_no_agree). wall-time: 3.54s, user-time: 0.08s, sys-time: 0.21s
Test (part2.concurrent_start): Concurrent starts
Pass (part2.concurrent_start). wall-time: 0.96s, user-time: 0.01s, sys-time: 0.03s
Test (part2.rejoin): Rejoin of partitioned leader
Pass (part2.rejoin). wall-time: 1.56s, user-time: 0.02s, sys-time: 0.16s
Test (part2.backup): Leader backs up quickly over incorrect follower logs
Pass (part2.backup). wall-time: 21.3s, user-time: 0.41s, sys-time: 1.82s
Test (part2.rpc_count): RPC counts aren't too high
Pass (part2.rpc_count). wall-time: 2.01s, user-time: 0.01s, sys-time: 0.07s
Pass 7/7 tests. wall-time: 34.26s, user-time: 0.54s, sys-time: 2.39s
```

Hints:

- Don't forget to initialize the states (e.g. lastApplied)
- Notice that the first log index is 1 instead of 0. To simplify the programming, you can append an empty log entry to the logs at the very beginning. And since the 'lastApplied' index starts from 0, the first empty log entry will never be applied to the state machine.
- Make sure your implementation is the same as the description of Figure 2 in the Raft paper.
- Do yourself a favor for future labs (especially for lab 3 and lab 4). Make your code clean and readable.
- Remember to use the mutex!
- Don't forget to implement the marshall and unmarshall method in `raft_protocol.cc` and `raft_protocol.h` (for the template class).
- The test cases may fail due to the bug from part 1.
- We have implemented the marshall and unmarshall function for the `command` in the tests. Therefore, you can marshall a `command` by `m << cmd` and unmarshall it by `u >> cmd`. Besides, you can also marshall/unmarshall a `std::vector` by `m << vec` / `u >> vec`.

Part 3 - Log Persistency

In this part, you will persist the states of a Raft node. Check Figure 2 in the Raft paper again, to figure out what should be persisted.

Recommended steps:

1. You should implement the class `raft_storeage` in `raft_storage.h` to persist the necessary states (e.g. logs). The test case will use the constructor `raft_storage(const std::string &file_dir)` to create a `raft_storage` object. Each raft node will

have its own file_dir to persist the states. And after a failure, the node will restore its storage via this dir.

2. You should use the `raft::storage` to persist the state, whenever they are changed.
3. And you should use the storage to restore the state when a Raft node is created.

You should pass the 6 test cases of this part. (5 points + 5 points + 5 points + 2 points + 2 points + 1 point)

```
% ./raft_test part3
Running 6 Tests ...
Test (part3.persist1): Basic persistence
Pass (part3.persist1). wall-time: 3.33s, user-time: 0s, sys-time: 0.03s
Test (part3.persist2): More persistence
Pass (part3.persist2). wall-time: 17.52s, user-time: 0.04s, sys-time: 0.25s
Test (part3.persist3): Partitioned leader and one follower crash, leader restarts
Pass (part3.persist3). wall-time: 2.73s, user-time: 0.03s, sys-time: 0.03s
Test (part3.figure8): Raft paper figure 8
Pass (part3.figure8). wall-time: 87.43s, user-time: 0.38s, sys-time: 1.22s
Test (part3.unreliable_agree): Agreement under unreliable network
Pass (part3.unreliable_agree). wall-time: 2.91s, user-time: 0.06s, sys-time: 0.17s
Test (part3.unreliable_figure_8): Raft paper Figure 8 under unreliable network
Pass (part3.unreliable_figure_8). wall-time: 26.15s, user-time: 0.02s, sys-time: 0.33s
Pass 6/6 tests. wall-time: 140.08s, user-time: 0.53s, sys-time: 2.03s
```

Hints:

- The test cases may fail due to the bugs from part 1 and part2.
- The network failure may cause the RPC library to print some errors, such as `rpcs::dispatch: unknown proc 3434.`. You don't need to worry about these errors since they won't cause your program to crash. But if the test crashes due to errors from the rpc library, please contact TA.
- To simplify your implementation, you don't have to consider the crash during the disk I/O. The test case won't crash your program during the I/O. For example, you don't have to make sure the atomicity of the state persists.
- You can use multiple files to persist different data (e.g. a file for metadata and the other for logs).
- To persist the command, you can use the `serialize` and `deserialize` interface of the `raft_command`.
- The log should be changed/persisted when the leader receives a new command or a follower receives an append_entries RPC.
- The metadata should be changed/persisted when the term changes or the node votes for someone.

Part 4 - Snapshot

In this part, you will implement the snapshot mechanism of the Raft algorithm. You can refer to Figure 13 in the Raft extended paper.

Notice: you don't need to partition the snapshot. You can send the whole snapshot in a single RPC.

Recommended steps:

1. Complete the classes and methods related to `raft::install_snapshot`.
2. Complete the method `raft::save_snapshot`.
3. Modify all the codes related to the log you have implemented before. (E.g. number of logs)
4. Restore the snapshot in the raft constructor.

You should pass the 3 test cases of this part. (2 points + 2 points + 1 points)

```
% ./raft_test part4
Running 3 Tests ...
Test (part4.basic_snapshot): Basic snapshot
Pass (part4.basic_snapshot). wall-time: 18.85s, user-time: 0.02s, sys-time: 0.1s
Test (part4.restore_snapshot): Restore snapshot after failure
Pass (part4.restore_snapshot). wall-time: 18.09s, user-time: 0.03s, sys-time: 0.11s
Test (part4.override_snapshot): Overrive snapshot
Pass (part4.override_snapshot). wall-time: 12.23s, user-time: 0s, sys-time: 0.07s
Pass 3/3 tests. wall-time: 49.17s, user-time: 0.05s, sys-time: 0.28s
```

Hints:

- You may skip this part and complete part 5 (which is much easier than this part) at first.
- This part may introduce many changes to your code base. So you'd better commit your codes before this part.
- To make the code clear, you can use two concepts for the log index: physical index (e.g. the index of the `std::vector`) and logical index (e.g. physical index + snapshot index).

Part 5 - Fault-tolerant Key-Value Store

In this part, you will use the library to build a fault-tolerant key-value store. You will implement a state machine that works as a single machine key-value store.

Recommended steps:

1. Complete the class `kv_state_machine, kv_command` in `raft_state_machine.h` and `raft_state_machine.cc`.

Notice: The command is executed asynchronously when applied to the state machine. Therefore, to get the result of the command, we provide a struct named `result` in the `kv_command`. You should fill this struct when applying the command. The usage should be like this:

```
std::unique_lock<std::mutex> lock(res->mtx); // you must use the lock to avoid contention.
// The value of these fields should follow the definition in `raft_state_machine.h` .
res->done = true; // don't forget to set this
res->succ = succ;
res->key = key;
res->value = value;
res->cv.notify_all(); // notify the caller
```

You should pass the 3 test cases of this part. (4 points + 4 points + 3 points)

```
% ./raft_test part5
Running 3 Tests ...
Test (part5.basic_kv): Basic key-value store
Pass (part5.basic_kv). wall-time: 1.15s, user-time: 0.03s, sys-time: 0.09s
Test (part5.persist_kv): Persist key-value store
Pass (part5.persist_kv). wall-time: 2.32s, user-time: 0.06s, sys-time: 0.1s
Test (part5.snapshot_kv): Persist key-value snapshot
Pass (part5.snapshot_kv). wall-time: 5.92s, user-time: 0.24s, sys-time: 0.32s
Pass 3/3 tests. wall-time: 9.39s, user-time: 0.33s, sys-time: 0.51s
```

Hints:

- You don't need to serialize/deserialize or marshall/unmarshall the `kv_command::res` field.

Grading

After you have implmented all the parts above, run the grading script:

```
./grade.sh
```

IMPORTANT: The grade scrip will run each test case many times. Once a test case failes, you will not get the score of that case. So, please make sure there are no concurrent bugs.

Handin Procedure

After all the above done:

```
% make handin
```

That should produce a file called lab3.tgz in the directory. Change the file name to your student id:

```
% mv lab3.tgz lab3_[your student id].tgz
```

Then upload lab3_[your student id].tgz file to Canvas before the deadline.

You'll receive full credits if your code passes the same tests that we gave you, when we run your code on our machines.