Project2 RaftKV

Raft is a consensus algorithm that is designed to be easy to understand. You can read material about the Raft itself at the Raft site, an interactive visualization of the Raft, and other resources, including the extended Raft paper.

In this project, you will implement a high available kv server based on raft, which needs you not only to implement the Raft algorithm but also use it practically, and bring you more challenges like managing raft's persisted state with `badger`, add flow control for snapshot message, etc.

The project has 3 parts you need to do, including:

- Implement the basic Raft algorithm
- Build a fault-tolerant KV server on top of Raft
- Add the support of raftlog GC and snapshot

Part A

The Code

In this part, you will implement the basic raft algorithm. The code you need to implement is under `raft/`. Inside `raft/`, there are some skeleton code and test cases waiting for you. The raft algorithm you're gonna implement here has a well-designed interface with the upper application. Moreover, it uses a logical clock (named tick here) to measure the election and heartbeat timeout instead of a physical clock. That is to say, do not set a timer in the Raft module itself, and the upper application is responsible to advance the logical clock by calling `RawNode.Tick()`. Apart from that, messages sending and receiving along with other things are processed asynchronously, it is also up to the upper application when to actually do these things (see below for more detail). For example, Raft will not block waiting on the response of any request message.

Before implementing, please checkout the hints for this part first. Also, you should take a rough look at the proto file `proto/proto/eraftpb.proto`. Raft sends and receives messages and related structs are defined there, you'll use them for the implementation. Note that, unlike Raft paper, it divides Heartbeat and AppendEntries into different messages to make the logic more clear.

This part can be broken down into 3 steps, including:

- Leader election
- Log replication
- Raw node interface

Implement the Raft algorithm

`raft.Raft` in `raft/raft.go` provides the core of the Raft algorithm including message handling, driving the logic clock, etc. For more implementation guides, please check `raft/doc.go` which contains an overview design and what these `MessageTypes` are responsible for.

Leader election

To implement leader election, you may want to start with `raft.Raft.tick()` which is used to advance the internal logical clock by a single tick and hence drive the election timeout or heartbeat timeout. You don't need to care about the message sending and receiving logic now. If you need to send out a message, just push it to `raft.Raft.msgs` and all messages the raft received will be passed to `raft.Raft.Step()`. The test code will get the messages from `raft.Raft.msgs` and pass response messages through `raft.Raft.Step()`. The `raft.Raft.Step()` is the entrance of message handling, you should handle messages like `MsgRequestVote`, `MsgHeartbeat` and their response. And please also implement test stub functions and get them called properly like `raft.Raft.becomeXXX` which is used to update the raft internal state when the raft's role changes.

You can run `make project2aa` to test the implementation and see some hints at the end of this part.

Log replication

To implement log replication, you may want to start with handling `MsgAppend` and `MsgAppendResponse` on both the sender and receiver sides. Checkout `raft.RaftLog` in `raft/log.go` which is a helper struct that helps you manage the raft log, in here you also need to interact with the upper application by the `Storage` interface defined in `raft/storage.go` to get the persisted data like log entries and snapshot.

You can run `make project2ab` to test the implementation and see some hints at the end of this part.

Implement the raw node interface

`raft.RawNode` in `raft/rawnode.go` is the interface we interact with the upper application, `raft.RawNode` contains `raft.Raft` and provide some wrapper functions like `RawNode.Tick()`and `RawNode.Step()`. It also provides `RawNode.Propose()` to let the upper application propose new raft logs.

Another important struct `Ready` is also defined here. When handling messages or advancing the logical clock, the `raft.Raft` may need to interact with the upper application, like:

- send messages to other peers
- save log entries to stable storage
- save hard state like the term, commit index, and vote to stable storage
- apply committed log entries to the state machine
- etc

But these interactions do not happen immediately, instead, they are encapsulated in `Ready` and returned by `RawNode.Ready()` to the upper application. It is up to the upper application when to call `RawNode.Ready()` and handle it. After handling the returned `Ready`, the upper application also needs to call some functions like `RawNode.Advance()` to update `raft.Raft`'s internal state like the applied index, stabled log index, etc.

You can run `make project2ac` to test the implementation and run `make project2a` to test the whole part A.

> Hints:
>
> - Add any state you need to `raft.Raft`, `raft.RaftLog`, `raft.RawNode` and message on `eraftpb.proto`

> - The tests assume that the first time start raft should have term 0
> - The tests assume that the newly elected leader should append a noop entry on its term
> - The tests assume that once the leader advances its commit index, it will broadcast the commit index by `MessageType_MsgAppend` messages.
> - The tests doesn't set term for the local messages, `MessageType_MsgHup`, `MessageType_MsgBeat` and `MessageType_MsgPropose`.
> - The log entries append are quite different between leader and non-leader, there are different sources, checking and handling, be careful with that.
> - Don't forget the election timeout should be different between peers.
> - Some wrapper functions in `rawnode.go` can implement with `raft.Step(local message)`
> - When starting a new raft, get the last stabled state from `Storage` to initialize `raft.Raft` and `raft.RaftLog`

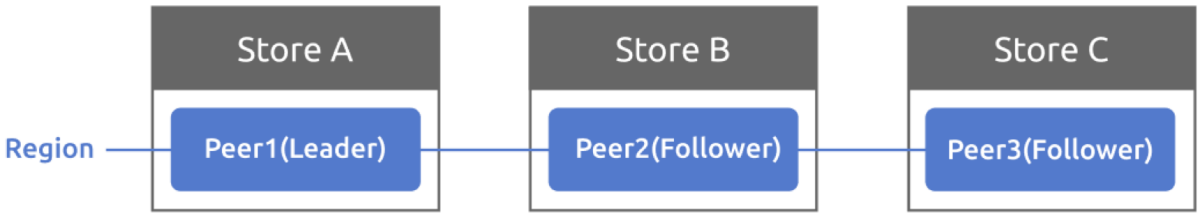
Part B

In this part, you will build a fault-tolerant key-value storage service using the Raft module implemented in part A. Your key/value service will be a replicated state machine, consisting of several key/value servers that use Raft for replication. Your key/value service should continue to process client requests as long as a majority of the servers are alive and can communicate, despite other failures or network partitions.

In project1 you have implemented a standalone kv server, so you should already be familiar with the kv server API and `Storage` interface.

Before introducing the code, you need to understand three terms first: `Store`, `Peer` and `Region` which are defined in `proto/proto/metapb.proto`.

- Store stands for an instance of tinykv-server
- Peer stands for a Raft node which is running on a Store
- Region is a collection of Peers, also called Raft group



For simplicity, there would be only one Peer on a Store and one Region in a cluster for project2. So you don't need to consider the range of Region now. Multiple regions will be further introduced in project3.

The Code

First, you should take a look at `RaftStorage` in `kv/storage/raft_storage/raft_server.go` which also implements the `Storage` interface. Unlike `StandaloneStorage` which directly writes to or reads from the underlying engine, it sends every write and read request to Raft first, and then does actual write and read to the underlying engine after Raft committed the request. Through this way, it can keep the consistency between multiple Stores.

`RaftStorage` creates a `Raftstore` to drive Raft. When calling the `Reader` or `Write` function, it actually sends a `RaftCmdRequest` defined in `proto/proto/raft_cmdpb.proto` with four basic command types(Get/Put/Delete/Snap) to raftstore by channel(the channel is `raftCh` of `raftWorker`) and returns the response after Raft commits and applies the command. The `kvrpc.Context` parameter of `Reader` and `Write` function is useful now, it carries the Region information from the perspective of the client and is passed as the header of `RaftCmdRequest`. The information might be incorrect or stale, so raftstore needs to check them and decides whether to propose the request.

Then, here comes the core of TinyKV — raftstore. The structure is a little complicated, read the TiKV reference to gain a better understanding of the design:

- https://pingcap.com/blog-cn/the-design-and-implementation-of-multi-raft/#raftstore (Chinese Version)
- https://pingcap.com/blog/design-and-implementation-of-multi-raft/#raftstore (English Version)

The entrance of raftstore is `Raftstore`, see `kv/raftstore/raftstore.go`. It starts some workers to handle specific tasks asynchronously, most of them aren't used now so you can just ignore them. All you need to focus on is `raftWorker`.(kv/raftstore/raft_worker.go)

The whole process is divided into two parts: raft worker polls `raftCh` to get the messages including base tick to drive Raft module and Raft commands to be proposed as Raft entries; it gets and handles ready from Raft module, including send raft messages, persist the state, apply the committed entries to the state machine. Once applied, return the response to clients.

Implement peer storage

Peer storage is what you interact with through the `Storage` interface in part A, but in addition to the raft log, peer storage also manages other persisted metadata which is very important to restore the consistent state machine after a restart. Moreover, there are three important states defined in `proto/proto/raft_serverpb.proto`:

- RaftLocalState: Used to store HardState of the current Raft and the last Log Index.
- RaftApplyState: Used to store the last Log index that Raft applies and some truncated Log information.
- RegionLocalState: Used to store Region information and the corresponding Peer state on this Store. Normal indicates that this Peer is normal and Tombstone shows that this Peer has been removed from Region and cannot join in Raft Group.

These states are stored in two badger instances: raftdb and kvdb:

- raftdb stores raft log and `RaftLocalState`
- kvdb stores key-value data in different column families, `RegionLocalState` and `RaftApplyState`. You can regard kvdb as the state machine mentioned in Raft paper

The format is as below and some helper functions are provided in `kv/raftstore/meta`, and set them to badger with `writebatch.SetMeta()`.

Key	KeyFormat	Value	DB
raft_log_key	0x01 0x02 region_id 0x01 log_idx	Entry	raft

Key	KeyFormat	Value	DB
raft_state_key	0x01 0x02 region_id 0x02	RaftLocalState	raft
apply_state_key	0x01 0x02 region_id 0x03	RaftApplyState	kv
region_state_key	0x01 0x03 region_id 0x01	RegionLocalState	kv

> You may wonder why TinyKV needs two badger instances. Actually, it can use only one badger to store both raft log and state machine data. Separating into two instances is just to be consistent with TiKV design.

These metadatas should be created and updated in `PeerStorage`. When creating PeerStorage, see `kv/raftstore/peer_storage.go`. It initializes RaftLocalState, RaftApplyState of this Peer, or gets the previous value from the underlying engine in the case of restart. Note that the value of both RAFT_INIT_LOG_TERM and RAFT_INIT_LOG_INDEX is 5 (as long as it's larger than 1) but not 0. The reason why not set it to 0 is to distinguish with the case that peer created passively after conf change. You may not quite understand it now, so just keep it in mind and the detail will be described in project3b when you are implementing conf change.

The code you need to implement in this part is only one function: `PeerStorage.SaveReadyState`, what this function does is to save the data in `raft.Ready` to badger, including append log entries and save the Raft hard state.

To append log entries, simply save all log entries at `raft.Ready.Entries` to raftdb and delete any previously appended log entries which will never be committed. Also, update the peer storage's `RaftLocalState` and save it to raftdb.

To save the hard state is also very easy, just update peer storage's `RaftLocalState.HardState` and save it to raftdb.

> Hints:
>
> - Use `WriteBatch` to save these states at once.
> - See other functions at `peer_storage.go` for how to read and write these states.
> - Set the environment variable LOG_LEVEL=debug which may help you in debugging, see also the all available log levels.

Implement Raft ready process

In project2 part A, you have built a tick-based Raft module. Now you need to write the outer process to drive it. Most of the code is already implemented under `kv/raftstore/peer_msg_handler.go` and `kv/raftstore/peer.go`. So you need to learn the code and finish the logic of `proposeRaftCommand` and `HandleRaftReady`. Here are some interpretations of the framework.

The Raft `RawNode` is already created with `PeerStorage` and stored in `peer`. In the raft worker, you can see that it takes the `peer` and wraps it by `peerMsgHandler`. The `peerMsgHandler` mainly has two functions: one is `HandleMsg` and the other is `HandleRaftReady`.

`HandleMsg` processes all the messages received from raftCh, including `MsgTypeTick` which calls `RawNode.Tick()` to drive the Raft, `MsgTypeRaftCmd` which wraps the request from clients and

`MsgTypeRaftMessage` which is the message transported between Raft peers. All the message types are defined in `kv/raftstore/message/msg.go`. You can check it for detail and some of them will be used in the following parts.

After the message is processed, the Raft node should have some state updates. So `HandleRaftReady` should get the ready from Raft module and do corresponding actions like persisting log entries, applying committed entries and sending raft messages to other peers through the network.

In a pseudocode, the raftstore uses Raft like:

```
for {
  select {
  case <-s.Ticker:
    Node.Tick()
  default:
    if Node.HasReady() {
      rd := Node.Ready()
      saveToStorage(rd.State, rd.Entries, rd.Snapshot)
      send(rd.Messages)
      for _, entry := range rd.CommittedEntries {
        process(entry)
      }
      s.Node.Advance(rd)
    }
}
```

After this the whole process of a read or write would be like this:

- Clients calls RPC RawGet/RawPut/RawDelete/RawScan
- RPC handler calls `RaftStorage` related method
- `RaftStorage` sends a Raft command request to raftstore, and waits for the response
- `RaftStore` proposes the Raft command request as a Raft log
- Raft module appends the log, and persist by `PeerStorage`
- Raft module commits the log
- Raft worker executes the Raft command when handing Raft ready, and returns the response by callback
- `RaftStorage` receive the response from callback and returns to RPC handler
- RPC handler does some actions and returns the RPC response to clients.

You should run `make project2b` to pass all the tests. The whole test is running a mock cluster including multiple TinyKV instances with a mock network. It performs some read and write operations and checks whether the return values are as expected.

To be noted, error handling is an important part of passing the test. You may have already noticed that there are some errors defined in `proto/proto/errorpb.proto` and the error is a field of the gRPC response. Also, the corresponding errors which implement the`error` interface are defined in `kv/raftstore/util/error.go`, so you can use them as a return value of functions.

These errors are mainly related to Region. So it is also a member of `RaftResponseHeader` of `RaftCmdResponse`. When proposing a request or applying a command, there may be some errors. If that, you should return the raft command response with the error, then the error will be further passed to gRPC response. You can use `BindRespError` provided in `kv/raftstore/cmd_resp.go` to convert these errors to errors defined in `errorpb.proto` when returning the response with an error.

In this stage, you may consider these errors, and others will be processed in project3:

- ErrNotLeader: the raft command is proposed on a follower. so use it to let the client try other peers.
- ErrStaleCommand: It may due to leader changes that some logs are not committed and overrided with new leaders' logs. But the client doesn't know that and is still waiting for the response. So you should return this to let the client knows and retries the command again.

> Hints:
>
> - `PeerStorage` implements the `Storage` interface of the Raft module, you should use the provided method `SaveReadyState()` to persist the Raft related states.
> - Use `WriteBatch` in `engine_util` to make multiple writes atomically, for example, you need to make sure to apply the committed entries and update the applied index in one write batch.
> - Use `Transport` to send raft messages to other peers, it's in the `GlobalContext`,
> - The server should not complete a get RPC if it is not part of a majority and does not has up-to-date data. You can just put the get operation into the raft log, or implement the optimization for read-only operations that is described in Section 8 in the Raft paper.
> - Do not forget to update and persist the apply state when applying the log entries.
> - You can apply the committed Raft log entries in an asynchronous way just like TiKV does. It's not necessary, though a big challenge to improve performance.
> - Record the callback of the command when proposing, and return the callback after applying.
> - For the snap command response, should set badger Txn to callback explicitly.
> - After 2A, some tests you may need to run them multiple times to find bugs

Part C

As things stand now with your code, it's not practical for a long-running server to remember the complete Raft log forever. Instead, the server will check the number of Raft log, and discard log entries exceeding the threshold from time to time.

In this part, you will implement the Snapshot handling based on the above two part implementation. Generally, Snapshot is just a raft message like AppendEntries used to replicate data to followers, what makes it different is its size, Snapshot contains the whole state machine data at some point of time, and to build and send such a big message at once will consume many resource and time, which may block the handling of other raft messages, to amortize this problem, Snapshot message will use an independent connect, and split the data into chunks to transport. That's the reason why there is a snapshot RPC API for TinyKV service. If you are interested in the detail of sending and receiving, check `snapRunner` and the reference https://pingcap.com/blog-cn/tikv-source-code-reading-10/

The Code

All you need to change is based on the code written in part A and part B.

Implement in Raft

Although we need some different handling for Snapshot messages, in the perspective of raft algorithm there should be no difference. See the definition of `eraftpb.Snapshot` in the proto file, the `data` field on `eraftpb.Snapshot` does not represent the actual state machine data, but some metadata is used by the upper application which you can ignore for now. When the leader needs to send a Snapshot message to a follower, it can call `Storage.Snapshot()` to get a `eraftpb.Snapshot`, then send the snapshot message like other raft messages. How the state machine data is actually built and sent are implemented by the raftstore, it will be introduced in the next step. You can assume that once `Storage.Snapshot()` returns successfully, it's safe for Raft leader to send the snapshot message to the follower, and follower should call `handleSnapshot` to handle it, which namely just restore the raft internal state like the term, commit index and membership information, etc, from the `eraftpb.SnapshotMetadata` in the message, after that, the procedure of snapshot handling is finished.

Implement in raftstore

In this step, you need to learn two more workers of raftstore — raftlog-gc worker and region worker.

Raftstore checks whether it needs to gc log from time to time based on the config `RaftLogGcCountLimit`, see `onRaftGcLogTick()`. If yes, it will propose a raft admin command `CompactLogRequest` which is wrapped in `RaftCmdRequest` just like four basic command types(Get/Put/Delete/Snap) implemented in project2 part B. Then you need to process this admin command when it's committed by Raft. But unlike Get/Put/Delete/Snap commands write or read state machine data, `CompactLogRequest` modifies metadata, namely updates the `RaftTruncatedState` which is in the `RaftApplyState`. After that, you should schedule a task to raftlog-gc worker by `ScheduleCompactLog`. Raftlog-gc worker will do the actual log deletion work asynchronously.

Then due to the log compaction, Raft module maybe needs to send a snapshot. `PeerStorage` implements `Storage.Snapshot()`. TinyKV generates snapshots and applies snapshots in the region worker. When calling `Snapshot()`, it actually sends a task `RegionTaskGen` to the region worker. The message handler of the region worker is located in `kv/raftstore/runner/region_task.go`. It scans the underlying engines to generate a snapshot, and sends snapshot metadata by channel. At the next time Raft calling `Snapshot`, it checks whether the snapshot generating is finished. If yes, Raft should send the snapshot message to other peers, and the snapshot sending and receiving work is handled by `kv/storage/raft_storage/snap_runner.go`. You don't need to dive into the details, only should know the snapshot message will be handled by `onRaftMsg` after the snapshot is received.

Then the snapshot will reflect in the next Raft ready, so the task you should do is to modify the raft ready process to handle the case of a snapshot. When you are sure to apply the snapshot, you can update the peer storage's memory state like `RaftLocalState`, `RaftApplyState`, and `RegionLocalState`. Also, don't forget to persist these states to kvdb and raftdb and remove stale state from kvdb and raftdb. Besides, you also need to update `PeerStorage.snapState` to `snap.SnapState_Applying` and send `runner.RegionTaskApply` task to region worker through `PeerStorage.regionSched` and wait until region worker finishes.

You should run `make project2c` to pass all the tests.