HW 6

Introduction

A key-value store is a type of database that implements a hash table abstraction: it stores mappings from *keys* to *values*. Popular key-value stores today such as Redis, memcached, and etcd are staples of production-level systems in industry today. These key-value stores are built in a distributed fashion, often with one or more leader and some number of followers. This design allows the database to scale past the capacity of just one node.

A key concept in distributed systems is *fault tolerance*: the property that system should be robust and reliable, able to continue operating despite the failure of some individual components. With distributed key-value stores, we must account for the possibility that some nodes will fail and crash over a long period of time due to common issues such as hardware failure or network partition. In the case of failures, we must ensure that we do not lose any data. To achieve this, distributed databases employ a strategy called *replication*, where the same key-value pair is stored on multiple followers instead of just one.

However, replication creates a problem for obtaining *consensus* - for example, a naive implementation supporting replication may end up in a state where one set of followers stores a value for some key which differs from that of another set of followers. In short, a database with consensus issues will lead to inconsistencies in the data. To solve this problem, we use a *consensus protocol* to make sure that all of our followers (or at least a majority), agree on the correct data. While there are many complex consensus protocols such as Paxos and Raft, our key-value store will use an older (and simpler!) protocol called *Two-Phase Commit (2PC)*.

Key-Value Store

Getting Started

To acquire the skeleton code, run `git pull staff master` in your personal repo and a new directory `hw6` should appear. The portion of the homework will be in `hw6/tpc`.

Installing `grpcio`

First, in your VM, install Python gRPC bindings with the following commands:

```
sudo apt -y update
sudo apt -y install python3-pip
sudo pip3 install grpcio grpcio-tools
```

Learning Go

The level at which you need to understand go to complete this topic is not very deep. But still, you should know how variable assignment, interfaces, structs, functions, conditionals, etc work. I recommend taking a look at GoByExample to learn syntax. You need to understand at least the following sections:

- "Hello World" through "Multiple Return Values"
- "Recursion" - "Channel Buffering"

Professor Culler wrote his own tutorial for learning Go just for this class. It does a very good job of explaining features of the language and I would **highly recommend** it.

Go Commands

`go test -mod vendor <dir>` : Runs the unit tests in `dir`. Leaving the directory empty runs it in the current directory. Replacing the directory with `./...` runs it in all sub directories.

`gofmt -l -w pkg/` : run this in the `hw6` directory to format all Go files in the homework. The auto-grader will refuse to run unless your Go files are formatted correctly.

`go build -o tpc -mod vendor main.go` : Run this command to build the KV store and output as a binary called `tpc` .

Your Tasks

In this homework, you will build a distributed key-value store with two operations: *GET* and *PUT*. Data will be replicated across multiple follower servers to ensure data integrity, and a single leader server will coordinate actions across these follower servers. All nodes in this key-value store will utilize the Two-Phase Commit protocol.

Multiple clients (users) will communicate with a single leader server according to the given Key-Value gRPC API. The leader will forward all GET requests to a random follower. For PUT requests, the leader should follow the Two Phase Commit (2PC) protocol to ensure that:

(1) operations are performed atomically across multiple follower servers
(2) backup data is consistent across multiple follower servers

The leaders and the followers communicate over a bi-directional gRPC stream. The leader sends a Leader message, which the follower will process and respond with a Response message.

The staff solution for parts A and B combined make the following changes:

```
journal/journal.go      |    3 ++
journal/journal_test.go |   32 ++++++++++++++++++++++++++
tpc/tpcfollower.go      |   55 ++++++++++++++++++++++++++++++++++++++++++++++++
tpc/tpcleader.go        |   55 ++++++++++++++++++++++++++++++++++++++++++++++++
4 files changed, 141 insertions(+), 4 deletions(-)
```

Getting Familiar

You may find it helpful to read through some of the pre existing code before you start implementing. Using Important Existing Code as a guide, we suggest that you read through:

- KVStore and Journal
- TPC Leader and TPC Follower
- MessageManager
- tpc.proto (`pkg/tpc/rpc`) and kv.proto (api)

TODO

Here is a list of things to do for this homework in the recommended order of completion at a high level - detailed specifications of Part A and Part B follow in the next pages.

Part A

- [ ] Read through the unit tests in `pkg/kvstore` and `pkg/journal` as well as the code in both directories. Then complete the unimplemented test in Journal.
- [ ] Using the unit test, identify and fix the bug in Journal that causes `Empty()` to fail. Then submit both your test and the bug fix to the autograder, which will let you know if you were successful.
- [ ] Read through Testing, Two Phase Commit, and Important Existing Code to get a good sense of the TPC components.
- [ ] Start on Follower TPC Handling (don't worry about journaling for persistence yet). The follower is a state machine that receives messages from the Leader and updates the internal state based on what messages it received. After this step you should be passing `TPC Follower Commit` and `TPC Follower Abort` tests on the autograder.
- [ ] Now work on the Leader TPC handling (again, don't worry about persistence just yet). The leader responds to API requests and issues TPC commands to the Follower. Once you implement this, you should be passing `TPC Leader Commit` and `TPC Leader Abort` along with the `E2E` tests.

Part B

- [ ] Implement the logic for how the follower state machine responds to retransmitted messages. After this you should be able to pass the retransmit tests.
- [ ] Think about where Journal entries need to be written and how the replay function should work for both the Follower and the Leader. We recommend you write to the journal upon receiving every message in the Follower and right before sending out every leader message in the Leader. At the end of this, you should be able to pass all TPC tests.

Part A

Unit Testing

Test Driven Development (TDD) is a crucial part of designing large systems and software in general. Golang has built-in testing features that we will take advantage of for unit testing. In the skeleton code, the FileJournal actually has a bug in its Empty function. The FileJournal is built on top of a file where each line is a journal entry. If a crash were to happen, the journal is read and its entries are replayed. The Empty function is periodically called at the end of the transaction of prevent the journal from growing unnecessarily large.

1. Write a test to test the functionality of `Empty` in the file `pkg/journal/journal_test.go`. You may want to look at the other test in same file and those in `pkg/kvstore/kvstore_test.go` to get a sense of how testing works in Go. This test should make sure that the log is actually emptied when Empty is called. *Hint: You may want to use an EntryIterator to verify.*

2. Now that this new test fails on the current code, use the output of your test to find and fix the bug in the `FileJournal`.

The Journal itself is not used in Part A, but fixing this bug and writing the test is worth points for this Part A. You can read more about the Journal interface in Important Existing Code. If you plan on completing Part B, you will need to implement this before you start.

Follower 2PC Handling

You should implement the 2PC follower to correctly handle normal GET and two-phase commit messages. Refer to Two Phase Commit for details on the implementation of two-phase commit logic. Refer to Testing for how to bypass the leader to communicate with followers directly.

Start by looking at the function `HandleMessage` in `pkg/tpc/tpcfollower.go`. This function is called every time the follower receives a message from the leader.

Based on what the message is, it will call either `vote` or `global`. You are responsible for filling in these two functions to update the follower state machine and return the

appropriate response.

For Part A, there is not a need to implement any journaling or the `replayJournal` function.

Leader 2PC Handling

You should implement the 2PC leader to correctly handle normal GET and PUT requests. Refer to Two Phase Commit for details on the implementation of two-phase commit logic. The leader needs to send Leader messages to all of its followers as well as process the results from the followers. Refer to Testing for how to mock a follower for the Leader to communicate with.

Start by looking at the function `Put` . This is the function where all Two Phase Commit transactions begin. The purpose of this function is to update the leader state machine and send the correct messages to the followers. This function will call `voteRequest` and `globalRequest` .

`voteRequest` and `globalRequest` are responsible for sending the correct messages to the followers and parsing the response. Both functions call `SendMessage` , which returns a channel where the responses will be placed. You can expect there will eventually be `numFollowers` responses.

For `voteRequest` , if any of the responses are an ABORT then the function should return an ABORT. For `globalRequest` , it doesn't matter what each follower returns since the only thing that can be returned is an ACK. Rather, you should block until you receive `numFollower` ACKs.

One point of confusion here is what the `SendMessage` function does. `SendMessage` takes a LeaderMsg and sends it to every follower the leader is connected to. It returns a Go channel of the responses and you can expect that there will eventually be `numFollowers` items in that channel.

Channels are a Go feature that can be considered a built-in bounded queue. If the retry argument is set to True, then `SendMessage` will resend the same message to a follower that fails to respond within a timeout or responds with an error. If it is set to False, then such a follower will be considered to have responded with an ABORT by default. This may be useful in implementing the behavior of the Vote phases versus a Global phase.

For Part A, there is not need to implement any journaling or the `replayJournal` function.

Part B

Retransmitted Messages

To begin part B, you will need to go back and add functionality that allows the 2PC follower to handle the cases where the 2PC follower gets a retransmission of a previously seen message. For example, if the follower gets two COMMIT or ABORT messages in a row.

2PC Crash Handling

Now that you have the Leader and Follower 2PC implemented, you should be passing all the basic 2PC tests on the autograder as well as the End to End tests. Ideally, you also pass the retransmit tests as well. This is a good indicator that your internal state is correct – something that is crucial to Part B.

In Part B, you will implement the `replayJournal` function for both the Follower and the Leader. This function will iterate through the Journal and replay all the actions that have been committed to the journal. At the end, you should be able to continue receiving messages from where you left off in the Journal.

The journal can be read using an `EntryIterator` which as the functions `HasNext` and `Next`. The journal itself can be modified with the functions `Append` and `Empty`. You must implement journaling in the follower's `vote` and `global` functions. In the leader, you must implement journaling in `Put`.

In both, you should implement the `replayJournal` function. For the follower, this function should update the internal state based on what entries were logged. For the leader, this function should either make a global COMMIT or ABORT based on what was last logged.

When implementing `replayJournal` for the leader, you may at one point need to pass a context to some helper functions. The context is a concept in Golang that adds timeouts and deadlines to requests and you don't really need to understand how its used. (Although if you are curious this video is great.) The context normally comes from the gRPC `Put` call, but when replaying the journal we no longer have a context to work with. Thus we need to make a new context, which you can do with `context.Background()`.

References

Two Phase Commit

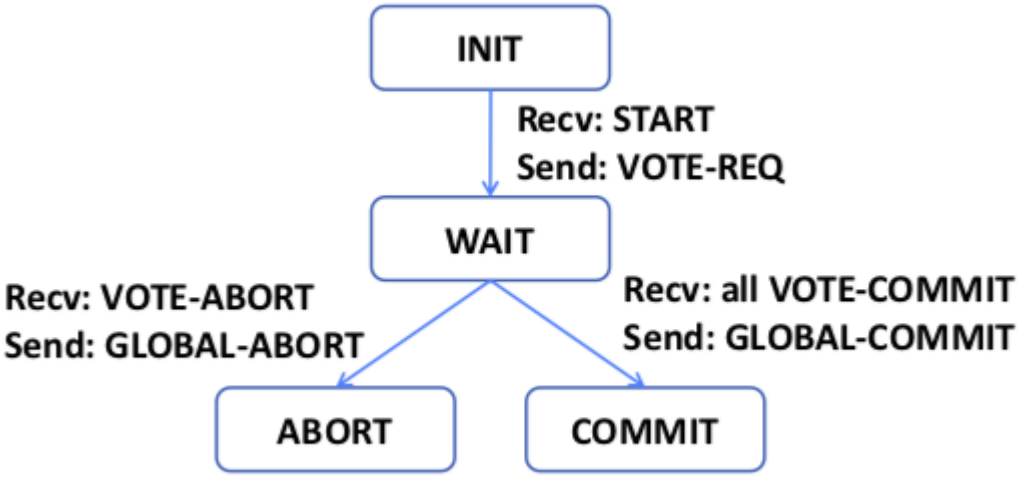
The high level ideas of two-phase commit:

(1) the leader blocks until it is sure that an operation is complete

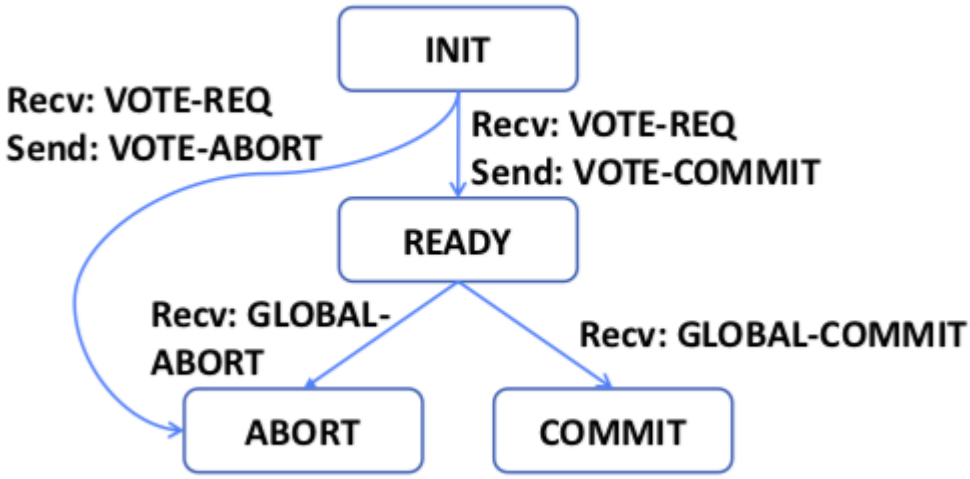
(2) an operation is only executed if every follower agrees to do so

1. Given an operation (i.e. PUT or DELETE), the leader (a.k.a. the global coordinator) will use a PREPARE phase where it asks every follower if it agrees to the operation.
2. Each of the followers can respond with a vote, which is either COMMIT (yes) or ABORT (no). The leader blocks until it has received a vote from every follower. If a follower does not respond within a timeout, the leader automatically assumes that the follower votes ABORT.
3. When the leader has received the vote from every follower, it will enter the COMMIT phase and send either a global COMMIT (if every follower voted to commit) or a global ABORT (if even a single follower voted no) to all of the followers.
4. The followers then execute the global command and respond with an ACK to acknowledge to the leader that it has completed the operation.
5. Only after the leader has received an ACK from every follower does it consider the operation complete and becomes available to handle the next command. If a follower does not respond within a timeout, the leader retransmits the global command until it does.

Two phase commit is implemented as a state machine, where communications between the leader and the followers result in state transitions. Below are the state diagrams for both the leader and the follower, respectively, which you will implement in this homework. Normally, completion of the global ABORT and COMMIT states should transition both the follower and the leader back into INIT.



State diagram for a 2PC Leader



State Diagram for a 2PC Follower

However in this homework, we skip the actual ABORT/COMMIT states as they only exist during execution of the global command and do not persist between commands. We instead only use `TPC_INIT` to designate a Follower ready to vote and `TPC_READY` to designate a Follower ready to execute a global command. The Leader does not require explicitly keeping track of what state it is on.

Important Existing Code

/vendor

The vendor folder stores all of the dependencies and libraries used by the Go program. All of the imports are not standard libraries will look for that library inside the vendor folder. You do not need to worry about code in here.

/test

The test folder contains Python code that can be used for writing and running some tests on the two phase commit participants locally. See Testing for more details.

/api

This folder contains the gRPC code that defines the Key Value store interface. It defines the service that implements Get and Put.

/pkg

This folder contains the source code for the Two Phase Commit KV Store.

/pkg/kvstore

This folder contains code that implements a file system based KV Store. It takes a path to a directory, and stores each entry as a separate file in that directory.

/pkg/journal

This folder contains code that implements a file based Journal. Each entry is stored as a separate line in the file. The journal has the following API:

`NewFileJournal(path string) (Journal, error)` : given a path, returns a Journal based on the file at that path. If no file exists, one is created. If a file does exist, the Journal is built from the entires already in that file.

`Size() int` : returns the numer of entries currently in the journal

`Empty()` : empties the journal by truncating the underlying file and resetting the in memory linked list

`Append(e Entry) error` : adds an entry to the journal by writing it to the underlying file and adding it to the in memory linked list.

`NewIterator() *EntryIterator` : returns an iterator over the in memory linked list of entries

/pkg/rpc

This folder contains the gRPC code that defines the Two Phase Commit interface. It defines the service that implements a bi-directional stream between the Leader and the Follower along with the structure of the messages they pass. Reading the `.proto` file here may be useful in figuring out what the Two Phase Commit API looks like.

/pkg/tpc

This folder contains the code for the Leader and Follower state machines. It also contains code for the gRPC framework for both the leader and the follower: a gRPC server that implements both the KV and TPC gRPC interfaces along with a framework that allows the leader and follower to send and receive TPC messages.

main.go

This file contains code that parses the command line inputs, creates the appropriate 2PC participant, and starts the gRPC server.

Testing

To test the KV store, you can build the program by running

```
go build -mod vendor -o tpc main.go
```

You will then have a binary called `tpc` (make sure you don't commit this). To test the application, you must start some number of 2PC followers and one 2PC leader. You can start a follower with the following command:

```
./tpc --tpc-follower --logtostderr \
    --journal-dir <path to journal file> \
    --kv-dir <path to key value directory> \
    --port <port> --name <name>
```

To start a leader, run the command:

```
./tpc --tpc-leader --logtostderr \
    --journal-dir <path to journal file> \
    --follower <url of follower> \
    --port <port> --name <name>
```

The URL of the follower is comprised of a hostname and a port such as `localhost:50001`. To connect multiple followers to a leader, you can set the `--follower` flag multiple times.

There is also the option to use a script to start a 2PC cluster automatically. Once you built the `tpc` binary, run the command `python3 tpc.py <num-followers>` to automatically start a cluster with the specified number of followers and a leader. Use Control-C to shut down the cluster when you are done. This cluster automatically exposes the leader at `localhost:50000`.

You can run end to end tests on the cluster with `python3 client.py <leader-url>`. It automatically connects to the leader and starts a simple shell where you can execute GET and PUT commands.

We also provide a framework for working with mocked followers and leaders. Take a look inside `client.py` to see how we use a fake follower and a fake leader in order to control the actions of each. We use this in order to run some tests, but if you want to use it you will need to write a python script that takes advantage of the mocks.

In the file `test.py`, we provide a framework very similar to out autograding framework. We also provide 3 test cases identical to the the cases used on the autograder, for you to use as an example to write your own tests. These examples also use the mock leader and mock followers, which is very useful for only testing either the follower or the leader in isolation.

Submitting

To submit your code, simply push to your personal repo's master branch. **Make sure you do not commit the `tpc` binary and you format your code before you push.** You can format your code by running `gofmt -l -w pkg/` in the `hw6` directory.

You need to manually run the tests for both part A and part B by going to the assignment on the autograder and pressing the `RUN AUTOGRADER` button.

We recommend that you start on this homework as soon as possible. The tests take a while to run, so towards the end you you wait a long time in queue before your build is executed!

Occasionally, you may get a test that fails with an output like:

```
...
File "/some/path/python3.7/subprocess.py", line 775, in __init__
    restore_signals, start_new_session)
File "some/path/python3.7/subprocess.py", line 1522, in _execute_child
    raise child_exception_type(errno_num, err_msg, err_filename)
OSError: [Errno 9] Bad file descriptor
```

We believe this is a bug with the subprocess library of Python itself. If you get this, simply restart another build as this bug only appears with some limited probability.