CS134 - Spring 2020

CS134 - Assignment 2: Primary/Backup Key/Value Service

Due: Thursday, April 30, 10:00PM PT

Introduction

In the MapReduce assignment, handling failures was relatively easy because the workers did not maintain state. The master did maintain state, but you didn't have to make it fault-tolerant. Assignment 2 is a first step towards fault tolerance for servers with state.

Collaboration Policy

You must write all the code you submit, except for code that we give you as part of the assignment. You are not allowed to look at anyone else's solution, and you are not allowed to look at code from previous years. You may discuss the assignments with other students, but you may not look at or copy each others' code. Please do not publish your code or make it available to future CS134 students -- for example, please do not make your code visible on GitHub (i.e., use a private repo).

Road map for assignments 2-4

In the next 3 assignments you will build several key/value services. The services support three RPCs: Put(key, value), Append(key, arg), and Get(key). The service maintains a simple database of key/value pairs. Put() replaces the value for a particular key in the database. Append(key, arg) appends arg to key's value. Get() fetches the current value for a key.

The assignments differ in the degree of fault tolerance and performance they provide for the key/value service:

- Assignment 2 uses primary/backup replication, assisted by a view service that decides which machines are alive. The view service allows the primary/backup service to work correctly in the presence of network partitions. The view service itself is not replicated, and is a single point of failure.
- Assignment 3 uses the Paxos protocol to replicate the key/value database with no single point of failure, and handles network partitions correctly. This key/value service is slower than a non-replicated key/value server would be, but is fault tolerant.
- Assignment 4 is a sharded key/value database, where each shard replicates its state using Paxos. This key/value service can perform Put/Get operations in parallel on different shards, allowing it to support applications such as MapReduce that can put a high load on a storage system. Assignment 4 also has a replicated configuration service, which tells the shards what key range they are responsible for. It can change the assignment of keys to shards, for example, in response to changing load. Assignment 4 has the core of a real-world design for thousands of servers.

In each assignment you will have to do substantial design. We give you a sketch of the overall design (and code for the utility functions), but you will have to flesh it out and nail down a complete protocol. The tests explore your protocol's handling of failure scenarios as well as general correctness and basic performance. You may need to redesign (and thus reimplement) your solutions in light of problems exposed by the tests; careful thought and planning may help you avoid too many redesign cycles. We don't give you a description of the test cases (other than the Go code). In the real world, you would have to come up with them yourself; so, we encourage you to take some time to understand exactly what they are testing (and potentially create your own test cases to focus on fixing issues that arise).

Setting up GitHub repository

For assignments 2-4, each team work in their own private repository in the *S20-CS134* GitHub organization. The TAs will create your repositories for you. Within 24 hours of a TA creating a private repository for your team, you will automatically receive an invitation to join the *S20-CS134* GitHub organization. Your team's repository name will be the same as the team name you listed in the submission form (barring naming conflicts).

Once you receive an invitation for a private repo, please follow the instructions below while replacing `${TEAM_NAME}` with your team's repository name:

- Access your team's repository: as soon as you accept the invitation, you should be able to see your team's repository either listed under the "Repositories" tab or by going to `https://github.com/S20-CS134/${TEAM_NAME}`.
- Mirror the *S20-CS134/assignments-2-4-skeleton* repository: similar to what you did in assignment 1, you

are going to duplicate the assignment 2 skeleton repository into your team's private repository. The assignment 2 skeleton repository can be found here: *S20-CS134/assignments-2-4-skeleton* . Now let's create a mirrored clone of this repository:

```
$ git clone --bare git@github.com:S20-CS134/assignments-2-4-skeleton.git
$ cd assignments-2-4-skeleton.git
$ git push --mirror git@github.com:S20-CS134/${TEAM_NAME}.git
$ cd ..
$ git clone git@github.com:S20-CS134/${TEAM_NAME}.git
$ rm -rf assignments-2-4-skeleton.git
$ cd ${TEAM_NAME}
```

After completing these steps, you should have all the required files for assignment 2 in your team's private repository. You can now start the assignment!

Overview of assignment 2

In this assignment you'll make a key/value service fault-tolerant using a form of primary/backup replication. In order to ensure that all parties (clients and servers) agree on which server is the primary, and which is the backup, we'll introduce a kind of master server, called the viewservice. The viewservice monitors whether each available server is alive or dead. If the current primary or backup becomes dead, the viewservice selects a server to replace it. A client checks with the viewservice to find the current primary. The servers cooperate with the viewservice to ensure that at most one primary is active at a time.

Your key/value service will allow replacement of failed servers. If the primary fails, the viewservice will promote the backup to be primary. If the backup fails, or is promoted, and there is an idle server available, the viewservice will cause it to be the backup. The primary will send its complete database to the new backup, and then send subsequent Puts to the backup to ensure that the backup's key/value database remains identical to the primary's.

It turns out the primary must send Gets as well as Puts to the backup (if there is one), and must wait for the backup to reply before responding to the client. This helps prevent two servers from acting as primary (a "split brain"). An example: S1 is the primary and S2 is the backup. The view service decides (incorrectly) that S1 is dead, and promotes S2 to be primary. If a client thinks S1 is still the primary and sends it an operation, S1 will forward the operation to S2, and S2 will reply with an error indicating that it is no longer the backup (assuming S2 obtained the new view from the viewservice). S1 can then return an error to the client indicating that S1 might no longer be the primary (reasoning that, since S2 rejected the operation, a new view must have been formed); the client can then ask the view service for the correct primary (S2) and send it the operation.

A failed key/value server may restart, but it will do so without a copy of the replicated data (i.e., the keys and values). That is, your key/value server will keep the data in memory, not on disk. One consequence of keeping data only in memory is that if there's no backup, and the primary fails, and then restarts, it cannot then act as primary.

Only RPC may be used for interaction between clients and servers, between different servers, and between different clients. For example, different instances of your server are not allowed to share Go variables or files.

The design outlined here has some fault-tolerance and performance limitations which make it too weak for real-world use:

- The view service is vulnerable to failures, since it's not replicated.
- The primary and backup must process operations one at a time, limiting their performance.
- A recovering server must copy a complete database of key/value pairs from the primary, which will be slow, even if the recovering server has an almost-up-to-date copy of the data already (e.g. only missed a few minutes of updates while its network connection was temporarily broken).
- The servers don't store the key/value database on disk, so they can't survive simultaneous crashes (e.g., a site-wide power failure).
- If a temporary problem prevents primary to backup communication, the system has only two remedies: change the view to eliminate the backup, or keep trying; neither performs well if such problems are frequent.
- If a primary fails before acknowledging the view in which it is primary, the view service cannot make progress---it will spin forever and not perform a view change.

We will address these limitations in later assignments by using better designs and protocols. This assignment will help you understand the problems that you'll solve in the succeeding assignments.

The primary/backup scheme in this assignment is not based on any published protocol. In fact, this assignment doesn't specify a complete protocol; you must work out the details.

Getting started

Assuming that you are in `${TEAM_NAME}` directory, running `test_test.go` in the `src/viewservice` should give the following errors:

```
$ export GOPATH=$(pwd)
$ cd src/viewservice
$ go test


2019/04/09 20:16:35 rpc.Register: method "GetRPCCount" has 1 input parameters; needs exactly three
2019/04/09 20:16:35 rpc.Register: method "Kill" has 1 input parameters; needs exactly three
Test: First primary ...
--- FAIL: Test1 (1.04s)
    test_test.go:13: wanted primary /var/tmp/824-501/viewserver-46150-1, got
FAIL
exit status 1
FAIL    viewservice 1.058s
```

Ignore the `method "GetRPCCount"` and `method "Kill"` error messages now and in the future. The tests fail because `viewservice/server.go` has empty RPC handlers.

You can run your code as stand-alone programs by building the source files (`src/main/viewd.go`, `src/main/pbd.go`, and `src/main/pbc.go`). Please see the comments in `src/main/pbc.go`.

Part A: The Viewservice

First you'll implement a viewservice and make sure it passes our tests; in Part B you'll build the key/value service. Your viewservice won't itself be replicated, so it will be relatively straightforward. Part B is *much* harder than part A, because the K/V service is replicated and you have to design much of the replication protocol.

The view service goes through a sequence of numbered *views*, each with a primary and (if possible) a backup. A view consists of a view number and the identity (network port name) of the view's primary and backup servers.

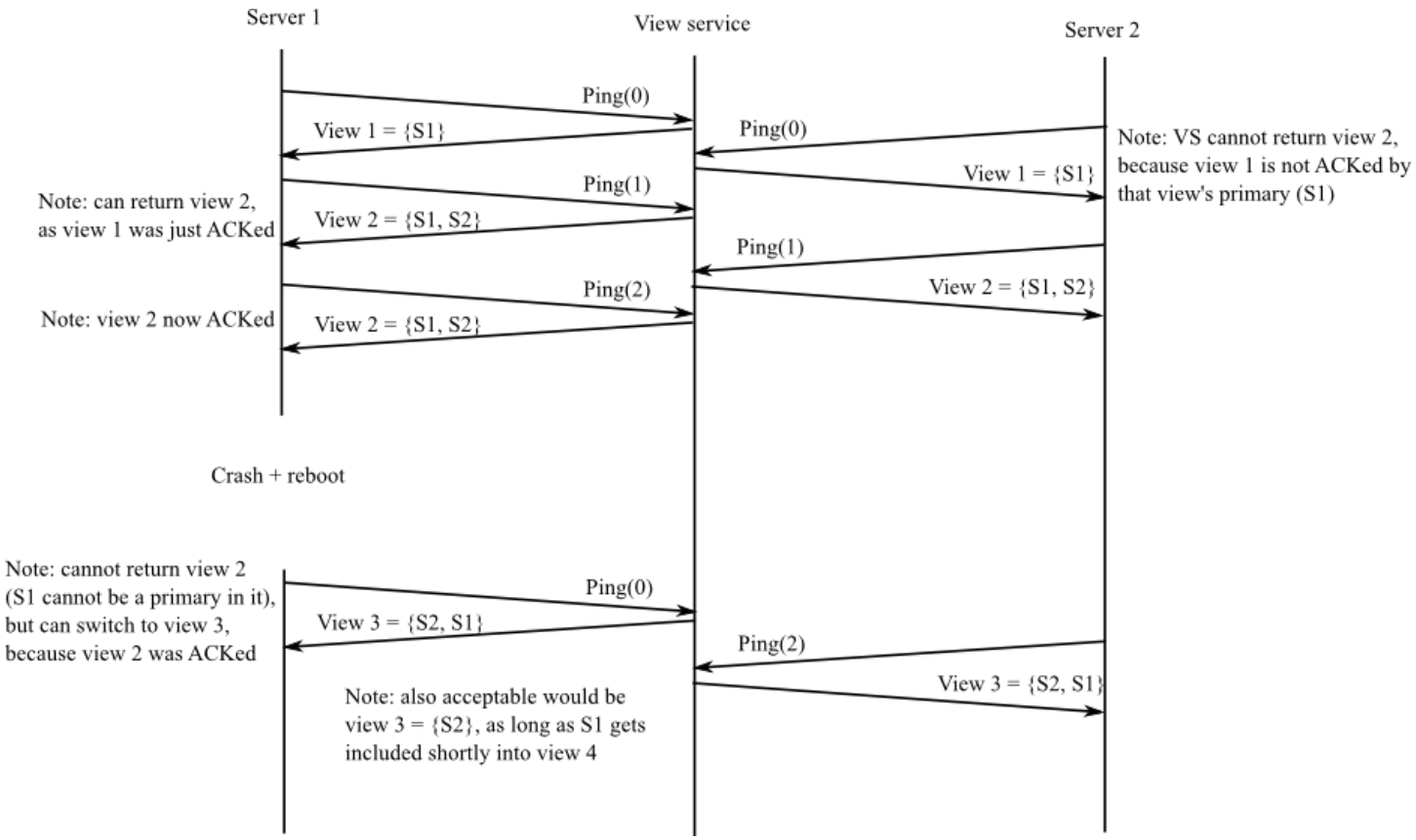
The primary in a view must always be either the primary or the backup of the previous view. This helps ensure that the key/value service's state is preserved. An exception: when the viewservice first starts, it should accept any server at all as the first primary. The backup in a view can be any server (other than the primary), or can be altogether missing if no server is available (represented by an empty string, `""`).

Each key/value server should send a Ping RPC once per `PingInterval` (see `viewservice/common.go`). The view service replies to the Ping with a description of the current view. A Ping lets the view service know that the key/value server is alive; informs the key/value server of the current view; and informs the view service of the most recent view that the key/value server knows about. If the viewservice doesn't receive a Ping from a server for `DeadPings` `PingIntervals`, the viewservice should consider the server to be dead. When a server re-starts after a crash, it should send one or more Pings with an argument of zero to inform the view service that it crashed.

The view service proceeds to a new view if it hasn't received recent Pings from both primary and backup, or if the primary or backup crashed and restarted, or if there is no backup and there is an idle server (a server that's been Pinging but is neither the primary nor the backup). But the view service must **not** change views (i.e., return a different view to callers) until the primary from the current view acknowledges that it is operating in the current view (by sending a Ping with the current view number). If the view service has not yet received an acknowledgment for the current view from the primary of the current view, the view service should not change views even if it thinks that the primary or backup has died. That is, the view service may not proceed from view X to view X+1 if it has not received a Ping(X) from the primary of view X.

The acknowledgment rule prevents the view service from getting more than one view ahead of the key/value servers. If the view service could get arbitrarily far ahead, then it would need a more complex design in which it kept a history of views, allowed key/value servers to ask about old views, and garbage-collected information about old views when appropriate. The downside of the acknowledgement rule is that if the primary fails before it acknowledges the view in which it is primary, then the view service cannot ever change views again.

An example sequence of view changes:



The above example is overspecified; for example, when the view server gets `Ping(1)` from S1 for the first time, it is also OK for it to return view 1, as long as it eventually switches to view 2 (which includes S2).

We provide you with a complete `client.go` and appropriate RPC definitions in `common.go`. Your job is to supply the needed code in `server.go`. When you're done, you should pass all the `viewservice` tests:

```
$ cd src/viewservice
$ go test
Test: First primary ...
  ... Passed
Test: First backup ...
  ... Passed
Test: Backup takes over if primary fails ...
  ... Passed
Test: Restarted server becomes backup ...
  ... Passed
Test: Idle third server becomes backup if primary fails ...
  ... Passed
Test: Restarted primary treated as dead ...
  ... Passed
Test: Viewserver waits for primary to ack view ...
  ... Passed
Test: Uninitialized server can't become primary ...
  ... Passed
PASS
ok      viewservice     7.457s
```

The above output omits some benign Go rpc errors.

Hint: you'll want to add field(s) to `ViewServer` in `server.go` in order to keep track of the most recent time at which the viewservice has heard a Ping from each server. Perhaps a `map` from server names to `time.Time`. You can find the current time with `time.Now()`.

Hint: add field(s) to `ViewServer` to keep track of the current view.

Hint: you'll need to keep track of whether the primary for the current view has acknowledged it (in PingArgs.Viewnum).

Hint: your viewservice needs to make periodic decisions, for example to promote the backup if the viewservice has missed `DeadPings` pings from the primary. Add this code to the `tick()` function, which is called once per `PingInterval`.

Hint: there may be more than two servers sending Pings. The extra ones (beyond primary and backup) are volunteering to be backup if needed.

Hint: the viewservice needs a way to detect that a primary or backup has failed and re-started. For example, the primary may crash and quickly restart without missing sending a single Ping.

Hint: study the test cases before you start programming. If you fail a test, you may have to look at the test code in `test_test.go` to figure out the failure scenario is.

The easiest way to track down bugs is to insert log.Printf() statements, collect the output in a file with `go test > out`, and then think about whether the output matches your understanding of how your code should behave.

Remember that the Go RPC server framework starts a new thread for each received RPC request. Thus if multiple RPCs arrive at the same time (from multiple clients), there may be multiple threads running concurrently in the server.

The tests kill a server by setting its `dead` flag. You must make sure that your server terminates when that flag is set (test it with `isdead()`), otherwise you may fail to complete the test cases.

Part B: The primary/backup key/value service

The primary/backup key/value server source is in `pbservice`. We supply you with part of a client interface in `pbservice/client.go`, and part of the server in `pbservice/server.go`. Clients use the service by creating a Clerk object (see `client.go`) and calling its methods, which send RPCs to the service.

Your key/value service should continue operating correctly as long as there has never been a time at which no server was alive. It should also operate correctly with partitions: a server that suffers temporary network failure without crashing, or can talk to some computers but not others. If your service is operating with just one server, it should be able to incorporate a recovered or idle server (as backup), so that it can then tolerate another server failure.

Correct operation means that calls to Clerk.Get(k) return the latest value set by a successful call to Clerk.Put(k,v) or Clerk.Append(k,v), or an empty string if the key has never seen either. All operations should provide at-most-once semantics.

You should assume that the viewservice never halts or crashes.

Your clients and servers may only communicate using RPC, and both clients and servers must send RPCs with the `call()` function in `client.go`.

It's crucial that only one primary be active at any given time. You should have a clear story worked out for why that's the case for your design. A danger: suppose in some view S1 is the primary; the viewservice changes views so that S2 is the primary; but S1 hasn't yet heard about the new view and thinks it is still primary. Then some clients might talk to S1, and others talk to S2, and not see each others' Put()s.

A server that isn't the active primary should either not respond to clients, or respond with an error: it should set GetReply.Err or PutReply.Err to something other than OK.

Clerk.Get(), Clerk.Put(), and Clerk.Append() should only return when they have completed the operation. That is, Put()/Append() should keep trying until they have updated the key/value database, and Clerk.Get() should keep trying until it has retrieved the current value for the key (if any). Your server must filter out the duplicate RPCs that these client re-tries will generate to ensure at-most-once semantics for operations. You can assume that each clerk has only one outstanding Put or Get. Think carefully about what the commit point is for a Put.

A server should not talk to the viewservice for every Put/Get it receives, since that would put the viewservice on the critical path for performance and fault-tolerance. Instead servers should Ping the viewservice periodically (in `pbservice/server.go`'s `tick()`) to learn about new views. Similarly, the client Clerk should not talk to the viewservice for every RPC it sends; instead, the Clerk should cache the current primary, and only talk to the viewservice when the current primary seems to be dead.

Part of your one-primary-at-a-time strategy should rely on the viewservice only promoting the backup from view *i* to be primary in view *i+1*. If the old primary from view *i* tries to handle a client request, it will forward it to its backup. If that backup hasn't heard about view *i+1*, then it's not acting as primary yet, so no harm done. If the backup has heard about view *i+1* and is acting as primary, it knows enough to reject the old primary's forwarded client requests.

You'll need to ensure that the backup sees every update to the key/value database, by a combination of the primary initializing it with the complete key/value database and forwarding subsequent client operations. Your primary should forward just the arguments to each `Append()` to the backup; do not forward the resulting value, which might be large.

The skeleton code for the key/value servers is in `src/pbservice`. It uses your viewservice, so make sure that your GOPATH is set correctly to where your `${TEAM_NAME}` directory is.

```
$ cd src/pbservice
$ go test -i
$ go test


2019/04/08 08:17:15 rpc.Register: method "GetRPCCount" has 1 input parameters; needs exactly three
2019/04/08 08:17:15 rpc.Register: method "Kill" has 1 input parameters; needs exactly three
Test: Single primary, no backup ...
--- FAIL: TestBasicFail (2.01s)
    test_test.go:54: first primary never formed view
```

```
2019/04/08 08:17:17 rpc.Register: method "GetRPCCount" has 1 input parameters; needs exactly three
2019/04/08 08:17:17 rpc.Register: method "Kill" has 1 input parameters; needs exactly three
Test: at-most-once Append; unreliable ...
--- FAIL: TestAtMostOnce (2.55s)
...
```

Here's a recommended plan of attack:

1. You should start by modifying `pbservice/server.go` to Ping the viewservice to find the current view. Do this in the `tick()` function. Once a server knows the current view, it knows if it is the primary, the backup, or neither.
2. Implement Get, Put, and Append handlers in `pbservice/server.go`; store keys and values in a `map[string]string`. If a key does not exist, Append should use an empty string for the previous value. Implement the `client.go` RPC stubs.
3. Modify your handlers so that the primary forwards updates to the backup.
4. When a server becomes the backup in a new view, the primary should send it the primary's complete key/value database.
5. Modify `client.go` so that clients keep re-trying until they get an answer. Make sure that you include enough information in `PutAppendArgs`, and `GetArgs` (see `common.go`) so that the key/value service can detect duplicates. Modify the key/value service to handle duplicates correctly.
6. Modify `client.go` to cope with a failed primary. If the current primary doesn't respond, or doesn't think it's the primary, have the client consult the viewservice (in case the primary has changed) and try again. Sleep for viewservice.PingInterval between re-tries to avoid burning up too much CPU time.

You're done if you can pass all the `pbservice` tests:

```
$ go test


Test: Single primary, no backup ...
  ... Passed
Test: Add a backup ...
  ... Passed
Test: Primary failure ...
  ... Passed
Test: Kill last server, new one should not be active ...
  ... Passed
Test: at-most-once Put; unreliable ...
  ... Passed
Test: Put() immediately after backup failure ...
  ... Passed
Test: Put() immediately after primary failure ...
  ... Passed
Test: Concurrent Put()s to the same key ...
  ... Passed
Test: Concurrent Put()s to the same key; unreliable ...
  ... Passed
Test: Repeated failures/restarts ...
  ... Put/Gets done ...
  ... Passed
Test: Repeated failures/restarts; unreliable ...
  ... Put/Gets done ...
  ... Passed
Test: Old primary does not serve Gets ...
  ... Passed
Test: Partitioned old primary does not complete Gets ...
  ... Passed
PASS
ok      pbservice       113.352s
```

You'll see some "method "GetRPCCount" has 1 input parameters" and "method "Kill" has 1 input parameters" complaints; ignore them.

Hint: you'll probably need to create new RPCs to forward client requests from primary to backup, since the backup should reject a direct client request but should accept a forwarded request.

Hint: you'll probably need to create new RPCs to handle the transfer of the complete key/value database from the primary to a new backup. You can send the whole database in one RPC (for example, include a `map[string]string` in the RPC arguments).

Hint: the state to filter duplicates must be replicated along with the key/value state.

Hint: the tester arranges for RPC replies to be lost in tests whose description includes "unreliable". This will cause RPCs to be executed by the receiver, but since the sender sees no reply, it cannot tell whether the server

executed the RPC.

Hint: some tests involve multiple clients trying to update the same key concurrently. You should make sure that the order of these updates is the same in both the primary and the backup

Hint: you may need to generate numbers that have a high probability of being unique. Try this:

```
import "crypto/rand"
import "math/big"
func nrand() int64 {
  max := big.NewInt(int64(1) << 62)
  bigx, _ := rand.Int(rand.Reader, max)
  x := bigx.Int64()
  return x
}
```

Hint: the tests kill a server by setting its `dead` flag. You must make sure that your server terminates correctly when that flag is set, otherwise you may fail to complete the test cases.

Hint: even if your viewserver passed all the tests in Part A, it may still have bugs that cause failures in Part B.

Hint: study the test cases before you start programming

Assignment Submission

To submit the assignment, please push your final code into your team's repository. Then fill out the submission form to turn in your assignment.

You will receive full credit if your code passes the `test_test.go` tests when we run your code on the SEASnet machines. You may use any number of your group's 4 late days for this assignment. To use a late day, remember to contact **both** TAs **ahead of time** (i.e., before 10pm PT on the assignment due date). Each used late day pushes the due date back by exactly 24 hours; for example, if you use 1 late day, the assignment will be due Friday, May 1 at 10pm. In the event that no late days are used, we will evaluate the code in your **last** submission prior to the deadline (i.e., using the commit hash listed in that submission).

Please post questions on Piazza.