Lab 2: Primary/Backup Key/Value Service

COSI 147A – Distributed Systems: Spring 2023
Part A Due: Friday February 10, 11:55 PM
Part B Due: Friday February 17, 11:55 PM

1 Introduction

There are situations that your lock service from Lab 1 does not handle correctly. For example, if a client thinks the primary is dead (due to a network failure or lost packets) and switches to the backup, but the primary is actually alive, the primary will miss some of the client's operations. Another problem with Lab 1 is that it does not cope with recovering servers: if the primary crashes, but is later repaired, it can't be reintegrated into the system, which means the system cannot tolerate any further failures.

In this lab you'll fix the above problems using a more sophisticated form of primary/backup replication. This time you'll build a **key/value storage service**. The service supports two RPCs: `Put(key, value)`, and `Get(key)`. The service maintains a simple database of key/value pairs. `Put()` updates the value for a particular key in the database. `Get()` fetches the current value for a key.

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 viewservice 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. `S1` can then return an error to the client indicating that `S1` is no longer the primary; the client can then ask the viewservice 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 in the lab has some fault-tolerance and performance limitations. The viewservice is vulnerable to failures because it is 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. 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.

The primary/backup scheme in this lab is not based on any published protocol. It is similar to the Flat Datacenter Storage (the viewservice is like FDS's metadata server, and the primary/backup servers are like FDS's tractservers), though FDS pays far more attention to performance. It's also a bit like a MongoDB replica set (though MongoDB selects the leader with a Paxos-like election). For a detailed description of a (different) primary-backup-like protocol, see Chain Replication paper (assigned later). Chain Replication has higher performance than this lab's design, though it assumes that the viewservice never declares a server dead when it is merely partitioned. See Harp paper (in auxiliary readings package) for a detailed treatment of high performance primary/backup and reconstruction of system state after various kinds of failures.

2 Collaboration Policy

You must write all the code you hand in, 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 other students** – for example, please do not make your code visible on Github or you will not receive credit for the assignment.

3 Software

We supply you with new skeleton code and new tests in `lab2/viewservice` and `lab2/pbservice`. You'll find the initial lab software on Latte. Access to a Unix-type OS is assumed; you can use the department's public workstations to run your code if needed. If you want to run your code on your machine and you encounter problems, please contact the TA. You can run your code as stand-alone programs using the source in the `cmd` directory; see the comments in `cmd/pbc/pbc.go`.

4 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.

The viewservice 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 it can be altogether missing if no server is available.

Each key/value server should send a `Ping` RPC once per `PingInterval` (see `lab2/viewservice/common.go`). A `Ping` lets the viewservice know that the key/value server is alive and informs the viewservice of the most recent view that the key/value server knows about. The `PingReply` of the viewservice informs the key/value server with a description of the current view.

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 restarts after a crash, it should send one or more `Pings` with an argument of zero to inform the viewservice that it crashed.

The viewservice proceeds to a new view when either it hasn't received a `Ping` from the primary or backup for `DeadPings` `PingIntervals`, 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 viewservice must not change views 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 viewservice has not yet received an acknowledgment for the current view from the primary of the current view, the viewservice should not change views even if it thinks that the primary or backup has died. (This implies another limitation of the system: If a primary dies before acknowledging a new view and fails to restart completely, the viewservice should stall.)

The acknowledgment rule prevents the viewservice from getting more than one view ahead of the key/value servers. If the viewservice 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.

Below is an example sequence of view changes:

```
Viewnum Primary Backup
______________________________________
0       none    none
Event: Server S1 sends Ping(0), which returns view 0
1       S1      none
Event: Server S2 sends Ping(0), which returns view 1
(no view change yet since S1 hasn't acked)
Event: Server S1 sends Ping(1), which returns view 1 or 2
2       S1      S2
Event: Server S1 sends Ping(2), which returns view 2
Event: Server S1 stops sending Pings
3       S2      none
Event: Server S2 sends Ping(3), which returns view 3
```

Note that this sequence omits events between the view changes. For example, ask yourself what else should need to happen before `S2` sends `Ping(3)`.

We provide you with a complete `lab2/viewservice/client.go` and appropriate RPC definitions in `lab2/viewservice/common.go`. Your job is to supply the needed code in `lab2/viewservice/server.go`. When you're done, you should pass all the `lab2/viewservice/test_test.go` tests:

```
$ cd lab2/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
```

4.1 Hints

1. 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()`.

2. Add field(s) to `ViewServer` to keep track of the current view.

3. You'll need to keep track of whether the primary for the current view has acknowledged it (using `PingArgs.Viewnum`).

4. 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`.

5. There may be more than two servers sending `Ping`s. The extra ones (beyond primary and backup) are volunteering to be backup if needed.

6. The viewservice needs a way to detect that a primary or backup has failed and restarted. For example, the primary may crash and quickly restart without missing sending a single `Ping`.

5 Part B: The Primary/Backup Key/Value 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 an empty string if the key has never been `Put()`. 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 `lab2/pbservice/common.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()` and `Clerk.Put()` should only return when they have completed the operation. That is, `Clerk.Put()` should keep trying until it has updated the key/value database, and `Clerk.Get()` should keep trying until if has retrieved the current value for the key (if any). Your server does not need to filter out the duplicate RPCs that these client retries will generate.

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 `lab2/pbservice/server.go`'s `tick()`) to learn about new views.

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 is 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 `Puts`.

The skeleton code for the key/value servers is in `lab2/pbservice`. Here's a recommended plan of attack:

1. You should start by modifying `lab2/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 `Put` and `Get` handlers in `pbservice/server.go`; store keys and values in a `map[string]string`.

3. Modify your `Put` handler 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. (**You should not send the entire database for every `Put` and `Get`**, however.)

5. Modify `lab2/pbservice/client.go` so that clients keep retrying until they get an answer.

6. If the current primary doesn't respond, or doesn't think it is the primary, have the client consult the viewservice (in case the primary has changed) and try again. Sleep for `viewservice.PingInterval` between retries to avoid burning up too much CPU time.

You're done if you can pass all the `lab2/pbservice/test_test.go` tests while not violating the restrictions above.

```
$ cd lab2/pbservice
$ 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: 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 ...
 ... Passed
Test: Repeated failures/restarts; unreliable ...
... Passed
Test: Old primary does not serve Gets ...
 ... Passed
Test: Partitioned old primary does not complete Gets ...
... Passed
PASS
ok pbservice 106.797s
```

5.1 Hints

1. 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.

2. You'll probably need to create new RPCs to transfer 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). However, this operation should be reserved for a view change, **not** for every `Put` and `Get`.

3. 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.

4. **Even if your viewserver passed all the tests in Part A, it may still have bugs that cause failures in Part B.**

6 Handin Procedure

Upload your code to Latte as a gzipped `tar` file by the deadlines at the top of the page. To do this, execute these commands:

```
$ cd ~/lab2
$ tar czvf lab2a-handin.tar.gz .
```

That should produce a file called `lab2a-handin.tar.gz`. And for Part B:

```
$ cd ~/lab2
$ tar czvf lab2b-handin.tar.gz .
```

You will receive full credit if your software passes the `test_test.go` tests and does not violate any of the conditions stated above. **We will run the tests over your code a minimum of twenty times**. There is no partial credit for tests: If a test case fails only once out of twenty times, it is still considered to have been failed.