Lab 3: Paxos-based Key/Value Service

COSI 147A – Distributed Systems: Spring 2025
Part A Due: Friday April 3, 11:55 PM
Part B Due: Tuesday April 23, 11:55 PM

1 Introduction

Your Lab 2 depends on a single master view server to pick the primary. If the view server is not available (crashes or has network problems), then your key/-value service won't work, even if both primary and backup are available. It also has the less critical defect that it copes with a server (primary or backup) that's briefly unavailable (e.g., due to a lost packet) by either blocking or declaring it dead; the latter is very expensive because it requires a complete key/value database transfer. In this lab you'll fix the above problems by using Paxos to manage the replication of a key/value store. You won't have anything corresponding to a master viewserver. Instead, a set of replicas will process all client requests in the same order, using Paxos to agree on the order. Paxos will get the agreement right even if some of the replicas are unavailable, or have unreliable network connections, or even if subsets of the replicas are isolated in their own network partitions. As long as Paxos can assemble a majority of replicas, it can process client operations. Replicas that were not in the majority can catch up later by asking Paxos for operations that they missed.

Your system will consist of the following players: clients, `kvpaxos` servers, and `Paxos` peers. Clients send `Put()`, `PutHash()`, and `Get()` RPCs to key/value servers (called `kvpaxos` servers). A client can send an RPC to any of the `kvpaxos` servers, and should retry by sending to a different server if there's a failure. Each `kvpaxos` server contains a replica of the key/value database; handlers for client `Get()` and `Put()` RPCs; and a Paxos peer. Paxos takes the form of a library that is included in each `kvpaxos` server. A `kvpaxos` server talks to its local `Paxos` peer (via method calls). The different Paxos peers talk to each other via RPC to achieve agreement on each operation.

Your Paxos library's interface supports an indefinite sequence of agreement "instances." The instances are numbered with sequence numbers. Each instance is either "decided" or not yet decided. A decided instance has a value. If an instance is decided, then all the Paxos peers that are aware that it is decided will agree on the same value for that instance. The Paxos library interface allows `kvpaxos` to suggest a value for an instance, and to find out whether an instance has been decided and (if so) what that instance's value is. Your `kvpaxos` servers will use Paxos to agree on the order in which client `Put()`s and `Get()`s execute. Each time a `kvpaxos` server receives a `Put()` or `Get()` RPC, it will use Paxos to cause some Paxos instance's value to be a description of that `Put()` or `Get()`. That instance's sequence number determines when the `Put()` or `Get()` executes relative to other `Put()`s and `Get()`s. In order to find the value to be returned by a `Get()`, `kvpaxos` should first apply all `Put()`s that are ordered before the `Get()` to its key/value database.

You should think of `kvpaxos` as using Paxos to implement a "log" of `Put/Get` operations. That is, each Paxos instance is a log element, and the order of operations in the log is the order in which all `kvpaxos` servers will apply the operations to their key/value databases. Paxos will ensure that the `kvpaxos` servers agree on this order. 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.

Your key/value service (`kvpaxos`) will support three RPCs: `Put(key, value)`, `PutHash(key, value)`, and `Get(key)`. The service will maintain a simple database of key/value pairs. `Put()` will update the value for a particular key in the database. `PutHash` will chain all values for a key together, which is useful for testing purposes; `PutHash` will store the hash(old value of the key in database, new supplied value) into database, and return the old value. `Get()` will fetch the current value for a key.

Your Paxos-based key/value storage system will have some limitations that would need to be fixed in order for it to be a serious system. It won't cope with crashes, since it stores neither the key/value database nor the Paxos state on disk. It requires the set of servers to be fixed, so one cannot replace old servers. Finally, it is slow: many Paxos messages are exchanged for each `Put()` and `Get()`. All of these problems can be fixed.

You should consult the Paxos lecture notes and the Paxos assigned reading. For a wider perspective, have a look at the Zookeeper paper.

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 `lab3/paxos` and `lab3/kvpaxos`. You'll find the initial lab software on Latte. You can use the department's public workstations to run your code. If you want to run your code on your machine and you encounter problems, please contact the TAs.

4 Part A: Paxos

First you'll implement a Paxos library. `paxos.go` contains descriptions of the methods you must implement. When you're done, you should pass all the tests in `paxos` directory:

```
$ cd $lab3/paxos
$ go test
Test: Single proposer ...
... Passed
Test: Many proposers, same value ...
... Passed
Test: Many proposers, different values ...
... Passed
Test: Out-of-order instances ...
... Passed
Test: Deaf proposer ...
... Passed
Test: Forgetting ...
... Passed
Test: Lots of forgetting ...
... Passed
Test: Paxos frees forgotten instance memory ...
... Passed
Test: Many instances ...
... Passed
Test: Minority proposal ignored ...
... Passed
Test: Many instances, unreliable RPC ...
... Passed
Test: No decision if partitioned ...
... Passed
Test: Decision in majority partition ...
... Passed
Test: All agree after full heal ...
... Passed
Test: One peer switches partitions ...
... Passed
Test: One peer switches partitions, unreliable ...
... Passed
Test: Many requests, changing partitions ...
... Passed
PASS
ok paxos 59.523s
$
```

Your implementation must support this interface:

```
px = paxos.Make(peers []string, me int)
px.Start(seq int, v interface{}) // start agreement on new instance
px.Status(seq int) (decided bool, v interface{}) // get info about an instance
px.Done(seq int) // ok to forget all instances <= seq
px.Max() int // highest instance seq known, or -1
px.Min() int // instances before this have been forgotten
```

An application calls `Make(peers, me)` to create a `Paxos` peer. The `peers` argument contains the ports of all the peers (including this one), and the `me` argument is the index of this peer in the `peers` array. `Start(seq,v)` asks Paxos to start agreement on instance `seq`, with proposed value `v`; `Start()` should return immediately, without waiting for agreement to complete. The application calls `Status(seq)` to find out whether the `Paxos` peer thinks the instance has reached agreement, and if so what the agreed value is. `Status()` should consult the local Paxos peer's state and return immediately; it should not communicate with other peers. The application may call `Status()` for old instances (but see the discussion of `Done()` below).

Your implementation should be able to make progress on agreement for multiple instances at the same time. That is, if application peers call `Start()` with different sequence numbers at about the same time, your implementation should run the Paxos protocol concurrently for all of them. You should not wait for agreement to complete for instance i before starting the protocol for instance $i + 1$. Each instance should have its own separate execution of the Paxos protocol.

A long-running Paxos-based server must forget about instances that are no longer needed and free the memory storing information about those instances. An instance is needed if the application still wants to be able to call `Status()` for that instance, or if another `Paxos` peer may not yet have reached agreement on that instance. Your Paxos should implement freeing of instances in the following way. When a particular peer application will no longer need to call `Status()` for any instance $\leq x$, it should call `Done(x)`. That `Paxos` peer can't yet discard the instances, since some other `Paxos` peer might not yet have agreed to the instance. So each `Paxos` peer should tell each other peer the highest `Done` argument supplied by its local application. Each `Paxos` peer will then have a `Done` value from each other peer. It should find the minimum, and discard all instances with sequence numbers $\leq$ that minimum. The `Min()` method returns this minimum sequence number plus one.

It's OK for your `Paxos` to piggyback the `Done` value in the agreement protocol packets; that is, it's OK for peer `P1` to only learn `P2`'s latest `Done` value the next time that `P2` sends an agreement message to `P1`. If `Start()` is called with a sequence number less than `Min()`, the `Start()` call should be ignored. If `Status()` is called with a sequence number less than `Min()`, `Status()` should return false (indicating no agreement). Here is the Paxos pseudocode (for a single instance) from the lecture:

```
proposer(v):
    while not decided:
        choose n, unique and higher than any n seen so far
        send prepare(n) to all servers including self
        if prepare_ok(n_a, v_a) from majority:
            v’ = v_a with highest n_a; choose own v otherwise
            send accept(n, v’) to all
            if accept_ok(n) from majority:
                send decided(v’) to all

acceptor’s state:
    n_p (highest prepare seen)
    n_a, v_a (highest accept seen)

acceptor’s prepare(n) handler:
    if n > n_p
        n_p = n
        reply prepare_ok(n_a, v_a)
    else
        reply prepare_reject

acceptor’s accept(n, v) handler:
    if n >= n_p
        n_p = n
        n_a = n
        v_a = v
        reply accept_ok(n)
    else
        reply accept_reject
```

4.1 Hints

Here’s a reasonable plan of attack:

1. Add elements to the `Paxos` struct in `paxos.go` to hold the state you’ll need, according to the lecture pseudocode. You’ll need to define a struct to hold information about each agreement instance.

2. Define RPC argument/reply type(s) for Paxos protocol messages, based on the lecture pseudocode. The RPCs must include the sequence number for the agreement instance to which they refer. Remember the field names in the RPC structures must start with capital letters.

3. Write a proposer function that drives the Paxos protocol for an instance, and RPC handlers that implement acceptors. Start a proposer function in its own thread for each instance, as needed (e.g., in `Start()`).

4. At this point you should be able to pass the first few tests.

5. Now implement forgetting.

6. More than one Paxos instance may be executing at a given time, and they may be `Start()`ed and/or decided out of order. (E.g., `seq` 10 may be decided before `seq` 5).

7. Remember that multiple application peers may call `Start()` on the same instance, perhaps with different proposed values. An application may even call `Start()` for an instance that has already been decided.

8. Think about how your paxos will forget (discard) information about old instances before you start writing code. Each Paxos peer will need to store instance information in some data structure that allows individual instance records to be deleted (so that the Go garbage collector can free/reuse the memory).

9. You do not need to write code to handle the situation where a Paxos peer needs to restart after a crash. If one of your Paxos peers crashes, it will never be restarted.

10. Have each Paxos peer start a thread per undecided instance whose job is to eventually drive the instance to agreement, by acting as a proposer.

11. A single Paxos peer may be acting simultaneously as acceptor and proposer for the same instance. Keep these two activities as separate as possible.

12. A proposer needs a way to choose a higher proposal number than any seen so far. This is a reasonable exception to the rule that proposer and acceptor should be separate. It may also be useful for the propose RPC handler to return the highest known proposal number if it rejects an RPC, to help the caller pick a higher one next time. The `px.me` value will be different in each Paxos peer, so you can use `px.me` to help ensure that proposal numbers are unique.

13. Figure out the minimum number of messages Paxos should use when reaching agreement in non-failure cases and make your implementation use that minimum.

14. The tester calls `Kill()` when it wants your Paxos to shut down; `Kill()` sets `px.dead`. You should check `px.dead` in any loops you have that might run for a while, and break out of the loop if `px.dead` is true. It's particularly important to do this any in any long-running threads you create.

5 Part B: Paxos-Based Key/Value Service

Now you'll build `kvpaxos`, a fault-tolerant key/value storage system. You'll modify `kvpaxos/client.go`, `kvpaxos/common.go`, and `kvpaxos/server.go`. Your `kvpaxos` replicas should stay identical; the only exception is that some replicas may lag others if they are not reachable. If a replica isn't reachable for a while, but then starts being reachable, it should eventually catch up (learn about operations that it missed). Your `kvpaxos` client code should try different replicas it knows about until one responds. A `kvpaxos` replica that is part of a majority of replicas that can all reach each other should be able to serve client requests.

Your storage system must provide sequential consistency to applications that use its client interface. That is, completed application calls to the `Clerk.Get()`, `Clerk.Put()`, and `Clerk.PutHash()` methods in `kvpaxos/client.go` must appear to have affected all replicas in the same order and have at-most-once semantics. A `Clerk.Get()` should see the value written by the most recent `Clerk.Put()` (in that order) to the same key. One consequence of this is that you must ensure that each application call to `Clerk.Put()` must appear in that order just once (i.e., write the key/value database just once), even though internally your `client.go` may have to send `Put()` and `PutHash()` RPCs multiple times until it finds a `kvpaxos` server replica that replies.

5.1 Hints

Here's a reasonable plan:

1. Fill in the `Op` struct in server.go with the "value" information that `kvpaxos` will use Paxos to agree on, for each client request. `Op` field names must start with capital letters. You should use `Op` structs as the agreed-on values – for example, you should pass `Op` structs to Paxos `Start()`. Go's RPC can marshall/unmarshall `Op` structs; the call to `gob.Register()` in `StartServer()` teaches it how.

2. Implement the `Put()` handler in `server.go`. It should enter a `Put Op` in the Paxos log (i.e., use Paxos to allocate a Paxos instance, whose value includes the key and value (so that other `kvpaxoses` know about the `Put()`)).

3. Implement a `Get()` handler. It should enter a `Get Op` in the Paxos log, and then "interpret" the the log before that point to make sure its key/value database reflects all recent `Put()`s.

4. Add code to cope with duplicate client `Put()`s – i.e., situations in which `Put()` in `client.go` sends the same request to multiple `kvpaxos` replicas. The `Put()`/`PutHash()` should execute just once.

5. Your server should try to assign the next available Paxos instance (sequence number) to each incoming client RPC. However, some other `kvpaxos` replica may also be trying to use that instance for a different client's operation. So the `kvpaxos` server has to be prepared to try different instances.

6. Your kvpaxos servers should not directly communicate; they should only interact with each other through the Paxos log.

7. As in Lab 2, you will need to uniquely identify client operations to ensure that they execute just once. Also as in Lab 2, you can assume that each clerk has only one outstanding `Put` or `Get`.

8. A kvpaxos server should not complete a Get() RPC if it is not part of a majority (so that it does not serve stale data). This means that each Get() (as well as each Put()) must involve Paxos agreement.

9. Don't forget to call the Paxos Done() method when a kvpaxos has processed an instance and will no longer need it or any previous instance.

Your code will need to wait for Paxos instances to complete agreement. The only way to do this is to periodically call Status(), sleeping between calls. How long to sleep? A good plan is to check quickly at first, and then more slowly:

```
to := 10 * time.Millisecond
for {
    decided, _ := kv.px.Status(seq)
    if decided {
        ...
        return
    }
    time.Sleep(to)
    if to < 10 * time.Second {
        to *= 2
    }
}
```

If one of your kvpaxos servers falls behind (i.e., did not participate in the agreement for some instance), it will later need to find out what (if anything) was agree to. A reasonable way to to this is to call Start(), which will either discover the previously agreed-to value, or cause agreement to happen. Think about what value would be reasonable to pass to Start() in this situation.

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 ~/lab3
$ tar czvf lab3a-handin.tar.gz .
```

That should produce a file called lab3a-handin.tar.gz. And for Part B:

```
$ cd ~/lab3
$ tar czvf lab3b-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 again 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.