6.824 - Spring 2015

6.824 Lab 3: Paxos-based Key/Value Service

Part A Due: Fri Feb 27 11:59pm

Part B Due: Fri Mar 13 11:59pm

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 view server. 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(), Append(), 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()/Append() 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, Append()s, and Get()s execute. Each time a kvpaxos server receives a Put()/Append()/Get() RPC, it will use Paxos to cause some Paxos instance's value to be a description of that operation. That instance's sequence number determines when the operation executes relative to other operations. In order to find the value to be returned by a Get(), kvpaxos should first apply all Put()s and Append()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/Append/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 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 client operation. 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 Chubby, Paxos Made Live, Spanner, Zookeeper, Harp, Viewstamped Replication, and Bolosky et al.

Collaboration Policy

You must write all the code you hand in for 6.824, 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 6.824 students -- for example, please do not make your code visible on github.

Software

Do a `git pull` to get the latest lab software. We supply you with new skeleton code and new tests in `src/paxos` and `src/kvpaxos`.

```
$ add 6.824
$ cd ~/6.824
$ git pull
...
$ cd src/paxos
$ go test
Single proposer: --- FAIL: TestBasic (5.02 seconds)
```

```
        test_test.go:48: too few decided; seq=0 ndecided=0 wanted=3
Forgetting: --- FAIL: TestForget (5.03 seconds)
        test_test.go:48: too few decided; seq=0 ndecided=0 wanted=6
...
$
```

Ignore the huge number of "has wrong number of ins" and "type Paxos has no exported methods" errors.

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 the `paxos` directory (after ignoring Go's many complaints):

```
$ cd ~/6.824/src/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) (fate Fate, 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 <= 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 <= 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 `Forgotten`.

Here is the Paxos pseudo-code (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, 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, 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
```

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 pseudo-code. 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 pseudo-code. 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.

Hint: 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).

Hint: in order to pass tests assuming unreliable network, your paxos should call the local acceptor through a function call rather than RPC.

Hint: 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.

Hint: 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 / re-use the memory).

Hint: you do not need to write code to handle the situation where a Paxos peer needs to re-start after a crash. If one of your Paxos peers crashes, it will never be re-started.

Hint: have each Paxos peer start a thread per un-decided instance whose job is to eventually drive the instance to agreement, by acting as a proposer.

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

Hint: 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.

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

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

Part B: Paxos-based Key/Value Server

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.Append() 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() or Clerk.Append() (in that order) to the same key. One consequence of this is that you must ensure that each application call to Clerk.Put() or Clerk.Append() 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

RPCs multiple times until it finds a kvpaxos server replica that replies.

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 PutAppend() handler in server.go. It should enter a Put or Append 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() or Append())). An Append Paxos log entry should contain the Append's arguments, but not the resulting value, since the result might be large.
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 requests, including situations where the client sends a request to one kvpaxos replica, times out waiting for a reply, and re-sends the request to a different replica. The client request should execute just once. Please make sure that your scheme for duplicate detection frees server memory quickly, for example by having the client tell the servers which RPCs it has heard a reply for. It's OK to piggyback this information on the next client request.

Hint: 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.

Hint: your kvpaxos servers should not directly communicate; they should only interact with each other through the Paxos log.

Hint: 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, Get, or Append.

Hint: 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() and Append()) must involve Paxos agreement.

Hint: 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.

Hint: 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 {
    status, _ := kv.px.Status(seq)
    if status == paxos.Decided{
      ...
      return
    }
    time.Sleep(to)
    if to < 10 * time.Second {
      to *= 2
    }
  }
```

Hint: 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.

Hint: When the test fails, check for gob error (e.g. "rpc: writing response: gob: type not registered for interface ...") in the log because go doesn't consider the error fatal, although it is fatal for the lab.

Handin procedure

Submit your code via the class's submission website, located here:

https://6824.scripts.mit.edu:444/submit/handin.py/

You may use your MIT Certificate or request an API key via email to log in for the first time. Your API key (XXX) is displayed once you logged in, which can be used to upload lab3 from the console as follows. For part A:

```
$ cd ~/6.824
$ echo XXX > api.key
$ make lab3a
```

And for part B:

```
$ cd ~/6.824
$ echo XXX > api.key
$ make lab3b
```

You can check the submission website to check if your submission is successful.

You will receive full credit if your software passes the `test_test.go` tests when we run your software on our machines. We will use the timestamp of your **last** submission for the purpose of calculating late days.

Please post questions on Piazza.