
Assignment 6: Paxos
Due: November 8

The goal of this programming assignment is to learn the knowledge about the distributed consensus and fault-tolerance of servers that is based upon active replications. Specifically, you have to implement a fault-tolerant key-value store which stores same copy of key-value pairs in multiple servers by using Paxos to achieve concensus between servers. Assume that there are n servers that keep the same key-value pairs. After completing parts A and B, your final system will accept the following requests from a client:

1. `Put` $\langle key \rangle$ $\langle value \rangle$ adds the key value pair into this store. key is type of String, value is type of Integer. Whenever a server receives a Put request from client, the server has to use Paxos to agree on the request to deliver.

2. `Get` $\langle key \rangle$ returns the value associated with the specific key. If no such key exists, return null.

You should consult the paper, Paxos Made Simple by Leslie Lamport. You can also consult MIT Distributed System lab3 http://nil.csail.mit.edu/6.824/2015/labs/lab-3.html. Finally, starter code and a JUnit test suite is provided for you. Please be sure to download these resources from canvas before beginning - they contain functionality needed for the grader.

Part A: Paxos(60%)

First you'll implement a Paxos library. Paxos.java contains descriptions of the methods you must implement. When you're done, you should pass all the tests in the PaxosTest.java. Your implementation must support this interface, which is given in the template:

```
px = new Paxos(int me, String[] peers, int[] ports)

px.Start(int seq, Object v);// start agreement on new instance

retStatus ret = px.Status(int seq);// get info about an instance

px.Done;// ok to forget all instances <= seq

int ret = px.Max();//highest instance seq known, or -1

int ret = px.Min();// instances before this have been forgotten
```

A server calls `Paxos(me, peers, ports)` to create a Paxos peer. The peers argument contains the hostnames of all the peers (including this one). The ports argument contains the ports of all the peers (including the local peer), 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 start a new thread using runnable interface, 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()`).

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 not equal to 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 less then that minimum. The `Min()` method returns this minimum sequence number plus one.

It's okay for your Paxos to piggyback the Done value in the agreement protocol packets; that is, it's okay 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.

To help you in your implementation, the Paxos pseudo-code (for a single instance) from the lecture is shown in Algorithm 1. Here's a reasonable plan of attack:

Add elements to the Paxos struct in paxos.java to hold the state you'll need, according to the above pseudo-code. You'll need to define a class to hold information about each agreement instance. Define RMI argument/reply type(s) for Paxos protocol messages, based on the lecture pseudo-code. The RMIs must include the sequence number for the agreement instance to which they refer. Write a proposer function that drives the Paxos protocol for an instance, and RMI handlers that implement acceptors. Start a proposer function in its own thread for each instance, as needed (e.g. in `Start()`). At this point you should be able to pass the first few tests. Now implement forgetting.

Some hints to keep in mind:

- 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).
- In order to pass tests assuming unreliable network, your paxos should call the local acceptor through a function call rather than RMI.
- 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.
- 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.

Algorithm 1 pseudo-code

```
1: proposer(v):
2: while not decided: do
3:    choose n, unique and higher than any n seen so far
4:    send prepare(n) to all servers including self
5:    if prepare_ok(n, n_a, v_a) from majority then
6:       v' = v_a with highest n_a; choose own v otherwise
7:       send accept(n, v') to all
8:       if accept_ok(n) from majority then
9:          send decided(v') to all
10:      end if
11:   end if
12: end while
13:
14: acceptor's state:
15: n_p (highest prepare seen)
16: n_a, v_a (highest accept seen)
17:
18: acceptor's prepare(n) handler:
19: if n > n_p then
20:   n_p = n
21:   reply prepare_ok(n, n_a, v_a)
22: else
23:   reply prepare_reject
24: end if
25:
26: acceptor's accept(n, v) handler:
27: if n >= n_p then
28:   n_p = n
29:   n_a = n
30:   v_a = v
31:   reply accept_ok(n)
32: else
33:   reply accept_reject
34: end if
```

- 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.
- 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.
- A single Paxos peer may be acting simultaneously as acceptor and proposer for the same instance. Keep these two activities as separate as possible.
- 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 RMI handler to return the highest known proposal number if it rejects an RMI, 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.

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

- 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(40%)

Now you'll build kvpaxos, a fault-tolerant key/value storage system. You'll modify kvpaxos/client.java, and kvpaxos/server.java.

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 `Client.Get()`, `Client.Put()` methods in kvpaxos/client.go must appear to have affected all replicas in the same order and have at-most-once semantics. `Client.Get()` should see the value written by the most recent `Client.Put()` (in that order) to the same key. One consequence of this is that you must ensure that each application call to `Client.Put()` must appear in that order just once (i.e., write the key/value database just once), even though internally your client.java may have to send RPCs multiple times until it finds a kvpaxos server replica that replies.

Here's a reasonable plan:

Fill in the Op class in server.java with the "value" information that kvpaxos will use Paxos to agree on, for each client request. You should use Op object as the agreed-on values – for example, you should pass Op object to Paxos `Start()`. We provided an implemented Op class that implements serializable so RMI can marshall/unmarshall Op objects. Implement the `Put` handler in server.java. 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()`). Implement a `Get()` handler. It should enter a Get Op in the Paxos log, and then "interpret" the log before that point to make sure its key/value database reflects all recent `Put()`s. 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.

Some hints to keep in mind:

- 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.

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

- You can assume that each client has only one outstanding `Put`, `Get`.
- A kvpaxos server should not complete a `Get()` RMI 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.
- 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:

```
public Op wait(int seq){
    int to = 10;
    while(true){
        Paxos.retStatus ret = this.px.Status(seq);
        if(ret.state == State.Decided){
            return Op.class.cast(ret.v);
        }
        try{
            Thread.sleep(to);
        } catch (Exception e){
            e.printStackTrace();
        }
        if( to < 1000){
            to = to * 2;
        }
    }
}
```

Handin procedure

You should not remove any source code from the template. The source code files such as paxos dir and kvpaxos dir should be under the same dir named as $EID1\_EID2$. Please do not remove the junit jar file. Then zip your $EID1\_EID2$ dir into $EID1\_EID2$.zip and submit through Canvas. Please follow this handin procedure, otherwise 10% will be deducted automatically. e.g.

```
zy6363_yh28827
├── paxos
│   └── Paxos.java
├── kvpaxos
│   ├── Server.java
│   └── Client.java
└── junit-4.12.jar
```