FINAL: QUESTION 2

Let's do that again with a slightly different crash/recover scenario for each process. Start with the following:

With all three members (mongod's) up and running, you should be fine; otherwise, delete your data directory, and, once again:

```
$ mongo --shell --port 27003 a.js
```

```
> ourinit() // you might need to wait a bit after this.
> // be sure 27003 is the primary.
> // use rs.stepDown() elsewhere if it isn't.
```

```
> db.foo.drop()
> db.foo.insert( { _id : 1}, {writeConcern : { w : 2 } } )
> db.foo.insert( { _id : 2}, {writeConcern : { w : 2 } } )
> db.foo.insert( { _id : 3}, {writeConcern : { w : 2 } } )
> var a = connect("localhost:27001/admin");
> a.shutdownServer()
> rs.status()
> db.foo.insert( { _id : 4 } )
> db.foo.insert( { _id : 5 } )
> db.foo.insert( { _id : 6 } )
```

Now this time, shut down the `mongod` on port 27003 (in addition to the other member being shut down by `testRollback()` already) before doing anything else. One way of doing this in Unix would be:

```
$ ps -A | grep mongod
$ # should see the 27003 and 27002 ones running (only)
$ ps ax | grep mongo | grep 27003 | awk '{print $1}' | xargs kill
$ # wait a little for the shutdown perhaps...then:
$ ps -A | grep mongod
$ # should get that just the arbiter is present…
```

Now restart just the 27001 member. Wait for it to get healthy -- check this with `rs.status()` in the shell. Then query

```
> db.foo.find()
```

Then add another document:

```
> db.foo.insert( { _id : "last" } )
```

After this, restart the third set member (`mongod` on port 27003). Wait for it to come online and enter a health state (secondary or primary).

Run (on any member -- try multiple if you like) :

```
> db.foo.find()
```

You should see a difference from problem 1 in the result above.

Question: Which of the following are true about mongodb's operation in these scenarios? Check all that apply.

- [ ] The MongoDB primary does not write to its datafiles until a majority acknowledgement comes back from the rest of the cluster. When 27003 was primary, it did not perform the last 3 writes.
- [ ] MongoDB preserves the order of writes in a collection in its consistency model. In this problem, 27003's oplog was effectively a "fork" and to preserve write ordering a rollback was necessary during 27003's recovery phase.
- [ ] When 27003 came back up, it transmitted its write ops that the other member had not yet seen so that it would also have them.

SUBMIT