ANSWER

Let's start this from scratch.

```
killall mongod
rm -r data
./a.sh
mongo --shell --port 27003 a.js
```

```
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()  // first server, if it's not down yet, should be down soon.
db.foo.insert( { _id : 4 } )
db.foo.insert( { _id : 5 } )
db.foo.insert( { _id : 6 } )
db.foo.find() // Let's see what we wrote.
exit
```

OK, now that that's done, let's shut down our server on port 27003. We can't issue shutdownServer in a way that takes down our replica set entirely (which this does), so we'll have to do it more manually. We'll find our process ID, put it into the variable 'processId', and throw away everything else in our query (which goes into the variable, 'otherStuff').

```
ps ax | grep mongo | grep 27003 | read processId otherStuff
kill $processId
```

OK, so both of our data bearing servers are now down. Our arbiter will not be of any help in retaining our data, as we'll see.

I can find the command to start up the other server, and launch it, by evaluating the appropriate command from a.js (assuming I haven't modified it):

```
cat a.sh | grep 27001 | read launchMongod
eval $launchMongod
mongo --port 27001
```

Once I've waited until the mongod at port 27001 is primary, I can do the following:

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

So I can see at this point that I've got only 4 of the documents. Let's see if I've got the rest when I bring up the other mongod.

```
cat a.sh | grep 27003 | grep z3 | read launchMongod
eval $launchMongod
mongo --port 27003
```

Let's see what we've got now.

```
rs.slaveOk()
db.foo.find() // This only shows 4 documents.
```

And there's all the information we need, if this information is combined with all that we learned from the course. Let's tackle the answers one by one:

- The MongoDB primary does not write to its data files until a majority acknowledgement comes back from the rest of the cluster. When 27003 was primary, it did not perform the last 3 writes.
    - This is false. We could see from the shell's response that the writes were written, and we could also see those documents when we did a find() after making the inserts. We also know that the default write concern is { w : 1 }, not { w : "majority" }, so this answer is doubly false.
- 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.
    - This is also false. When we checked for those writes on 27003 after it came back up, they were nowhere to be found, so this one can't be correct. We also should know from the lectures and documentation (if we checked it) that this isn't what happens.
- 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.

- This is correct. Since we had writes that were unknown to the `mongod` at port 27001, we had to stash those writes somewhere when the mongod at port 27003 came back up (and we'll see where in the next problem's answer). We can see from the state of the replica set when all servers were back up that its behavior was consistent with this answer, and we would have to add our knowledge of rollbacks (and when they're triggered) from the chapter or the documentation in order to know that this has happened. Regardless this is the one correct choice from the problem.