



*Department of Electrical Engineering and Computer Science*

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

**6.824 Distributed System Engineering: Spring 2016**

# Exam I

Write your name on this cover sheet. If you tear out any sheets, please write your name on them. You have 80 minutes to complete this quiz.

Some questions may be much harder than others. Read them all through first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, write down any assumptions you make. Write neatly. In order to receive full credit you must answer each question as precisely as possible.

You may use class notes, papers, and lab material. You may read them on your laptop, but you are not allowed to use any network. For example, you may not look at web sites or communicate with anyone.

<b>I (14)</b>	<b>II (7)</b>	<b>III (7)</b>	<b>IV (21)</b>	<b>V (14)</b>	<b>VI (14)</b>	<b>VII (21)</b>	<b>VIII (2)</b>	<b>Total (100)</b>

**Name:**

## I Lab 1

Ben Bitdiddle is writing a simplified version of the `schedule()` function for the MapReduce lab. This function takes in an array of Tasks for a particular map or reduce phase and sends them to available workers. The function should return once all of the tasks in this phase have been completed.

```
1 func (mr *Master) schedule(tasks []Task) {
2     done := 0
3
4     for _, task := range tasks {
5         go func(task Task) {
6
7             // Get an available worker and give them the task to run.
8             worker = <-mr.registerChannel
9             call(worker, "Worker.DoTask", task)
10
11             // Tell the master this task is done.
12             done++
13
14             // Reregister the worker as available.
15             mr.registerChannel <- worker
16
17         }(task)
18     }
19
20     // Wait for all of the jobs to return.
21     for {
22         if done == len(tasks) {
23             break
24         }
25     }
26
27     return
28 }
```

Ben notices that `schedule()` sometimes never finishes. However, he can see from looking at the output files that the workers have completed all of the tasks in the phase.

1. [7 points]: Identify a race in Ben's code that can cause `schedule()` to never return.

After fixing the race in the previous question, Ben's code passes all of the MapReduce tests. Now he hopes to make his implementation faster by returning an available worker to `mr.registerChannel` as soon as possible, that is, *before* incrementing `done`. He implements this by moving line 15 so it comes right after line 9. However, he notices that with this new plan `schedule()` never returns.

2. [7 points]: Why does this happen?

## II GFS

Alice uses GFS (as described by Ghemawat *et al.*) to store the output of her MapReduce job. One task of the MapReduce job uses a single `write` call to write MapReduce records to a pre-existing “out.txt” file. That is, her job overwrites “out.txt”, which was created by a previous run of her job (she doesn’t care about any of the old content). Unfortunately, the GFS write call returns an error.

A check application runs on another machine while the MapReduce job is attempting to write. The check application opens “out.txt”, reads it, and closes the file. To Alice’s surprise the check application reads the bytes Alice’s MapReduce job tried to write to “out.txt”, even though the write reported an error to the MapReduce job. If she runs the check application again, however, it reads the old bytes.

3. [7 points]: Describe a scenario in which this can happen.

### III VM-FT

After reading the VM-FT paper (by Scales *et al.*), Ben explores running the GFS master on VM-FT. He reasons that, without modifying the existing GFS master code, he can avoid the master being a single point of failure. Unfortunately, he finds that replicating the master with VM-FT results in much slower performance than with an unreplicated master: even GFS operations that merely fetch data from the master take significantly longer with the master running on VM-FT.

4. [7 points]: Explain what causes this decrease in performance.

## IV Lab 2

Ben is having trouble getting his Lab 2 to work. Here's the part of his code that sends out AppendEntries RPCs when a server acts as leader:

```
1 for {
2   // ... omitted code that waits either for a new client
3   // request or for a heartbeat interval.
4
5   for i, p := range rf.peers {
6     if i == rf.me {
7       continue
8     }
9     rf.mu.Lock()
10    if rf.state != LEADER {
11      rf.mu.Unlock()
12      break
13    }
14    n := rf.nextIndex[i]
15    args := AppendEntriesArgs{
16      // ...
17      Term: rf.currentTerm,
18      PrevLogIndex: n-1,
19      PrevLogTerm: rf.log[n-1].Term,
20      Entries: make([]LogEntry, len(rf.log[n:])),
21    }
22    copy(args.Entries, rf.log[n:])
23    rf.mu.Unlock()
24
25    var reply AppendEntriesReply
26    ok := p.Call("Raft.AppendEntries", args, &reply)
27    if ok && reply.Success {
28      rf.mu.Lock()
29      rf.nextIndex[i] = len(rf.log)
30      rf.matchIndex[i] = rf.nextIndex[i] - 1
31      rf.mu.Unlock()
32    }
33  }
34  // update commitIndex based on rf.matchIndex
35  // ...
36 }
```

**5. [7 points]:** Ben's implementation often fails tests because it commits too slowly. Describe a change to his code that will speed it up significantly when there are many Raft servers.

**6. [7 points]:** Ben discovers that his servers often commit different commands for the same index in the log. He discovers that this happens when the winner of an election was the leader in a previous term; in that situation the leader's matchIndex entries sometimes contain inexplicably large values shortly after the election, even though Ben's implementation sets nextIndex and matchIndex entries to zero when a server is elected leader. Explain how the code above can cause what Ben sees.

**7. [7 points]:** After fixing the problem identified in the previous question, Ben finds that a leader will sometimes commit a log entry that has not been accepted by a majority of servers. He finds that this is because the leader sets `matchIndex` too high after hearing a response from an `AppendEntries` RPC. Explain how Ben's code for updating `nextIndex` and `matchIndex` is wrong.

## V Zookeeper

In Zookeeper (as described by Hunt *et al.*), any server can respond to a read operation such as `GetData`. In lab 3, read operations such as `Get` are served by the Raft leader after committing the `Get` operation in Raft log. As a result, read operations in Zookeeper are fast and read operations in lab 3 are slow.

**8. [7 points]:** What does Zookeeper sacrifice compared to lab 3 by allowing each server to respond to a read operation? (Briefly explain your answer.)

A Zookeeper client  $C_1$  calls `setData("z", 27)` at server  $S_1$  to change the value of znode  $z$  to 27 from its previous value of 0. No further `setData` calls happen after that one call. Another client  $C_2$  calls `getData("z")` at server  $S_1$  and reads 27. Now  $S_1$  fails and client  $C_2$  switches to server  $S_2$ .

**9. [7 points]:** If client  $C_2$  calls `getData("z")` at server  $S_2$  does Zookeeper guarantee that the client will observe 27 rather than 0? (Briefly explain your answer.)

## VI FaRM

You're building a simple auction system using FaRM to store the bids (see the paper “No compromises: distributed transactions with consistency, availability, and performance”). Your application supports just one auction, with just two bidders. You store the two current bids in FaRM, in the `bids[]` array. Each user can bid any number of times with the `bid()` function, which returns a boolean indicating whether the bid is the highest so far. The function's `who` argument is 0 or 1, and indicates which user is bidding. The return value from `commit_transaction()` is true if the FaRM commit protocol committed the transaction, and false if it aborted.

```
// each user's bid is a distinct FaRM object.
type BidObject struct {
    value int
    // other stuff...
}

// bids are initialized to zero.
var bids [2]BidObject
bids[0].value = 0
bids[1].value = 0

bid(who, amount):
    start_transaction()
    if amount > bids[0].value && amount > bids[1].value:
        ret = true
        bids[who].value = amount
    else:
        ret = false
    if commit_transaction():
        return ret
    else:
        return false
```

Each `bids[]` array element is a separate FaRM object, with its own FaRM lock and version number.

**10. [7 points]:** Suppose the bids array starts out containing zeroes, and there are calls to `bid(0,10)` and `bid(1,20)` on different machines at roughly the same time (and no other activity). Could the `bid(1,20)` call return false? Explain your answer in terms of the commit protocol in the paper's Figure 4.

**11. [7 points]:** You can view FaRM's `VALIDATE` as an optimization of `LOCK` for objects that a transaction only reads. Suppose FaRM didn't have this optimization – suppose one were to eliminate the `VALIDATE` phase from Figure 4, and instead use `LOCK` for objects that are just read as well as for objects that are written. Explain why this would likely hurt performance.



**12. [7 points]:** For a sequentially-consistent shared-memory system please list *all* possible outputs from Processor 3. (The stores to  $x$  and  $y$  complete as indicated by the time line. For example, by the time processor 2 updates  $y$ ,  $x$  has been updated.)

Consider the same program running on TreadMarks (as described by Keleher *et al.*):

**13. [7 points]:** For TreadMarks please list *all* possible outputs from Processor 3.

Consider the following slight modification: Processor 3 prints only  $x$  by calling  $px$  and Processor 4 prints only  $y$  by calling  $py$ .

```
int x = 0;
int y = 0;

int f() {
    acquire(x_lock); // answer
    x = 10;
    release(x_lock); // answer
}
```

```
int g() {
    acquire(y_lock); // answer
    y = 11;
    release(y_lock); // answer
}
```

```
int px() {
    acquire(x_lock); // answer
    print x;
    release(x_lock); // answer
}
```

```
int py() {
    acquire(y_lock); // answer
    print y;
    release(y_lock); // answer
}
```

```
Processor 1:    f();
Processor 2:                g();
Processor 3:                px();
Processor 4:                py();
```

**14. [7 points]:** Annotate  $f$ ,  $g$ ,  $p_x$ , and  $p_y$  in the above figure so that the program on TreadMarks generates the same result as on a sequentially-consistent memory, but minimizes communication overhead. (Briefly justify your annotations.)

**VIII 6.824**

**15. [1 points]:** Lab 2 turned out to be much harder than we anticipated. If you ran into significant difficulties, what were they?

**16. [0.5 points]:** Which papers should we definitely keep for future years?

**17. [0.5 points]:** Is there any paper that you think we should delete?

**End of Exam I**