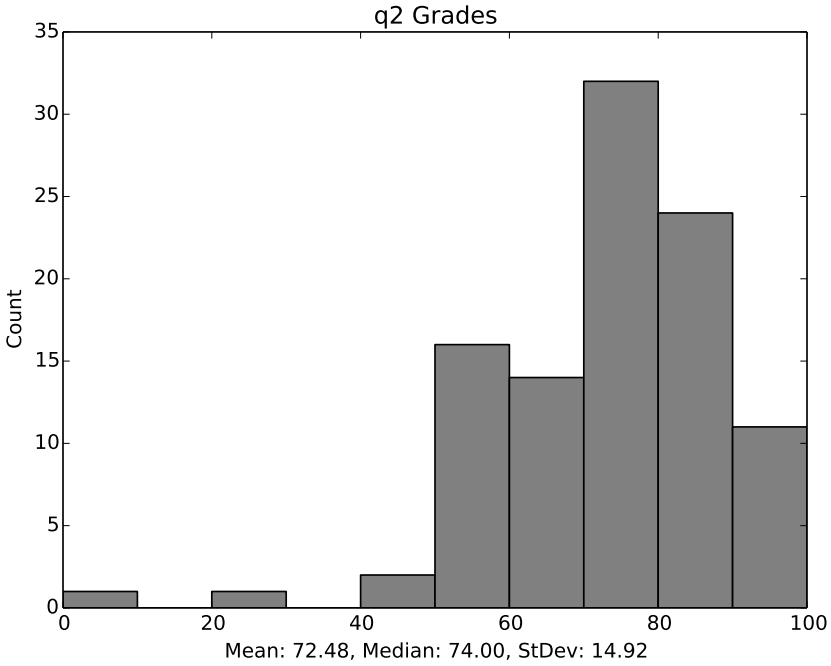
Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.824 Spring 2014

Quiz II Solutions



Histogram of grade distribution

I Memory models

Ben writes the following program in a variant of Go that guarantees a sequentially consistent memory model:

```
var a string = "hello"
var done bool

func main() {
  done = false;
  go func() {
    a = "world";
    done = true
  }()
  for !done {
    /* do nothing: spin until done */
  }
  print(a)
}
```

Ben's hope is that this program prints "world" after the go routine finishes.

1. [4 points]:

If the memory model is sequentially consistent, as in IVY, what is the output of this program? (Explain briefly why.)

Answer: world

The actual Go memory model is as follows:

> A read r of a variable v is allowed to observe a write w to v if both of the following hold:
>
> **A.** r does not happen before w.
>
> **B.** There is no other write w' to v that happens after w but before r.
>
> To guarantee that a read r of a variable v observes a particular write w to v, ensure that w is the only write r is allowed to observe. That is, r is guaranteed to observe w if both of the following hold:
>
> **A.** w happens before r.
>
> **B.** Any other write to the shared variable v either happens before w or after r.
>
> This pair of conditions is stronger than the first pair; it requires that there are no other writes happening concurrently with w or r.
>
> Within a single goroutine, there is no concurrency, so the two definitions are equivalent: a read r observes the value written by the most recent write w to v. When multiple goroutines access a shared variable v, they must use synchronization events to establish happens-before conditions that ensure reads observe the desired writes.

2. [4 points]:

If Ben compiles and runs the same program as before in the real Go memory model, what are all of the possible outcomes that Ben can observe when compiling and running this program? (Briefly explain.)

Answer: Several outcomes:

- "world", since the rules allow all writes to be immediately visible.
- "hello", since the rules do not require writes to one variable to be visible even if writes to another variable are.
- doesn't terminate, since there's no happens-before ordering between `done = true` and the `for !done` loop.

3. [2 points]: If the outcomes are the same under IVY and Go, why are they the same? If they are not the same, why are they different?

Answer: Different: IVY provides sequential consistency, and Go has a more relaxed memory model.

4. [4 points]:

Imagine the program from the previous question runs on TreadMarks, with the go routine running on a different machine. What are the possible outcomes? Briefly explain why.

Answer: The only outcome is the program not terminating (spinning forever), because the updates are never sent.

5. [4 points]:

How would you change the program to ensure that the desired behavior (outputting "world") happens under TreadMarks?

Answer: Update the shared variables `a` and `done` while holding a lock, and read them while holding a lock. Alternatively (not intended but still correct), just simplify the program: `func main() { print("world"); }`

II Ficus

Consider a single file f and 4 hosts: H1, H2, H3, and H4. f has a version vector associated with it with 4 entries, one for each host. Assume the initial version vector is [0, 0, 0, 0].

Let's say we have the following update and synchronization pattern:

```
H1: f = 1     ->H2                                    -> H4
H2:                    f = 2        -> H4
H3:
H4:                                          cat f          cat f
```

6. [4 points]: What will be the output of the two cat statements of f on H4?

Answer: 2, 2

7. [4 points]: What is the version of f at H4 at the first "cat"?

Answer: [1, 1, 0, 0]

8. [4 points]: What is the version of f that H1 sends to H4 when synchronizing?

Answer: [1, 0, 0, 0]

Consider the following update and synchronization pattern:

```
H1: f = 1    ->H2                          f = 3   -> H4
H2:                  f = 2       -> H4
H3:
H4:                                        cat f          cat f
```

9. [4 points]: What will be the output of the two cat statements of f on H4? Explain using version vectors.

Answer: "2", then some kind of error due to a conflict for the second cat. Version vector from H1 cannot be ordered with H4's version vector, when H1 syncs with H4.

III Bayou

In Bayou each update has a time stamp that consists of three components: a commit sequence number (CSN), a local time stamp, and a node identifier. The commit sequence number may be unknown for an update if a host hasn't synchronized with the central server. Consider the following scenario with a shared calendar, 3 hosts and a central server (*S*):

```
H1:             "10am Bob"  ->H2
H2:                                    "11am Alice"  -> S
H3: "10am Ben"   -> S
```

Assume that the local time stamps start at 0 and central server's CSN at 0. Both local time stamps and CSNs are bumped prior to assigning a value to a new update. For example, H1's calendar update is stamped with [-, 1, H1]. Further, assume that when nodes *A* and *B* synchronize, each node's clock becomes the max of the two node clocks before the sync.

10. [4 points]:

What is the time stamp for H2's update to the calendar before it synchronizes with *S*? Explain briefly why.

Answer: [-, 2, H2]. H2 updates its local time stamp counter after H1 syncs with its.

11. [4 points]:

What are the time stamp for H1 and H2's update to the calendar after synchronizing with *S*? Briefly explain why.

Answer: [2, 1, H1], [3, 2, H2]. H3 sneaks in first, so it gets the first CSN.

12. [4 points]:

After the above events, H4 syncs with H2 and then with H3. What is the displayed content of the shared calendar on H4 for 10am and 11am, in two cases: first, after syncing with H2, and second, after syncing with H3? Explain briefly why.

Answer: For both cases, the answer is the same, because H3 has no new updates relative to H2. The specific answer depends on how conflicts are resolved. If the conflicting update gets shifted to the next time slot, then the answer is (for both): "10am Ben" and "11am Bob" (and then "Noon Alice"). If the conflicting update is shifted to some other time slot, then the answer is (for both): "10am Ben" and "11am Alice".

13. [4 points]:

When the authors of Bayou developed that system, Paxos was not yet widely known or appreciated in the broader community. Now that you are all Paxos experts, do you think Bayou should be re-designed around the idea of using Paxos to assign CSNs across all of the nodes, instead of relying on a single server? Explain your reasoning.

Answer: Bad idea. Nodes may be offline, might never get a majority.

IV PNUTS

14. [8 points]:

Consider the following traces of invocations and responses made by a single client process in PNUTS. For each, say whether a client could observe this behavior on PNUTS (by marking True), or a client could never observe this on PNUTS (by marking False). Assume there are no failures (either servers or network). Assume there can be other concurrent clients.

A. True / False

Read-latest("foo") → ("xx", 3.7)
Read-any("foo") → ("xx", 3.5)

Answer: True. First goes to master in another data center. Second goes to a stale copy in local data center.

B. True / False

Read-any("foo") → ("xx", 3.7)
Read-any("foo") → ("xx", 3.5)

Answer: False. Local storage unit never rolls back, and client always talks to local data center when using read-any.

C. True / False

Read-latest("foo") → ("xx", 3.7)
Read-any("foo") → ("xx", 3.5)
Read-critical("foo", 3.6) → ("xx", 3.6)

Answer: True. First goes to master in another data center. Second goes to local data center, sees stale data. Third goes to local data center again, which by now received another update bumping it to version 3.6.

D. True / False

Read-latest("foo") → ("xx", 3.7)
Read-critical("foo", 3.6) → ("xx", 3.6)
Read-any("foo") → ("xx", 3.5)

Answer: False. Since read-latest returned 3.7, we know master is at version 3.7. Since read-critical returned 3.6, we know it went to the local data center, and not the master; and moreover, the local data center has version 3.6. At this point, read-any cannot return 3.5, because the local data center must have 3.6.

E. True / False

Read-any("foo") → ("xx", 3.5)
Write("foo", "yy") → 3.7
Read-any("foo") → ("xx", 3.6)

Answer: True. The first read went to the local data center. The write went to a master in another data center. The last read went to the local data center again, which by now received another update that was in flight at some point since the first read.

F. True / False

Read-latest("foo") → ("xx", 3.5)
Test-and-set-write("foo", 3.5, "yy") → false (failed)
Read-latest("foo") → ("yy", 3.6)

Answer: True. The test-and-set failed but some other client wrote “yy”.

For the next trace, assume that no other clients are active aside from the client whose trace is shown (but still no failures):

G. True / False

```
Read-latest("foo")        → ("xx", 3.5)
    Write("foo", "yy") → 3.6
 Read-any("foo")          → ("yy", 3.6)
    Write("foo", "zz") → 3.7
 Read-any("foo")          → ("yy", 3.6)
```

Answer: True. The write went to another data center, and in the first case it propagated to the local DC in time, but in the second case it didn't propagate to the local DC in time for read-any.

H. True / False

```
Read-latest("foo")        → ("xx", 3.1)
    Write("foo", "yy") → 3.2
          ...          → ...
    Write("foo", "yy") → 3.9
 Read-any("foo")          → ("yy", 3.9)
    Write("foo", "zz") → 3.10
 Read-any("foo")          → ("yy", 3.9)
```

Answer: False. After 3 writes, PNUTS migrates the record to the local data center. Thus, the last write must be visible to the last read-any.

15. [8 points]:

Alyssa P. Hacker is writing a Twitter-like application on top of PNUTS. When a user posts a new message, the application sends an email to the user's friends. Assume that each user has a single PNUTS record, storing a list of all messages from that user. Assume that the user's friends are stored elsewhere. The application has two core functions: `post(username, friendlist, msg)`, used by a user `username` with friends in `friendlist` to post message `msg`, and `view(username)`, used by anyone to view the list of messages from `username`.

Sketch out the pseudocode for these two functions below, trying to achieve the lowest latency whenever possible. Assume you have access to a function that sends email. You can modify the function signatures (arguments or return values) as needed, as long as you describe the semantics of your new function signature. The `post` function must send an email message pointing to a URL, which corresponds to some invocation of `view`; the email cannot contain the posted message in the email body.

Pseudocode:

Answer:

```
post(username, friendlist, msg):
  list, ver = read-any(username)
  while not test-and-set-write(username, ver, list + [msg]):
    list, ver = read-latest(username)
  for u in friendlist:
    email(u, make_post_url(username, ver+1))

view(username, ver):
  return read-critical(username, ver)
```

V Bitcoin

Ben Bitdiddle is writing his own implementation of a Bitcoin node. He is worried the blockchain grows without bound, so instead of storing the entire blockchain, his client stores:

- The last block in the blockchain, `last`, and
- A list of transactions that have not been spent yet, `unspent`.

When Ben's node receives a new block, it performs all of the standard checks that a normal Bitcoin node performs: it verifies that the new block chains to `last`, verifies that all transactions listed in the new block are legitimate (i.e., each of them chains to a previously unspent transaction listed in `unspent`), verifies that the new block hash is less than the target, etc.

Ben's node then updates the state accordingly, setting `last` to the new block and updating `unspent` (removing transactions that have been spent, and adding new unspent transactions that are unspent).

16. [4 points]:

Is there a situation where Ben's node can be tricked into accepting a transaction (e.g., double-spending) that wouldn't be accepted by a standard Bitcoin node?

Answer: We accepted either "No" or a well-described problem that arises when an adversary crafts two transactions (double-spending), gets lucky creating two blocks around the same time, one containing each of the transactions, feeds one block to Ben and one block to the rest of the Bitcoin network. At this point, since Ben has trouble switching between forks (see next question), Ben might never learn that his transaction was not widely accepted; the adversary can keep (slowly) mining blocks on Ben's fork, to eventually convince Ben that his transaction is valid (i.e., 5 more blocks have appeared after it).

17. [4 points]:

Are there any other problematic situations that might arise with Ben's node that wouldn't arise with a standard Bitcoin node?

Answer: Yes: Ben's node is unable to resolve forks without re-downloading the whole blockchain.

VI Lab 4

18. [10 points]:

Lab 4 includes the following hint:

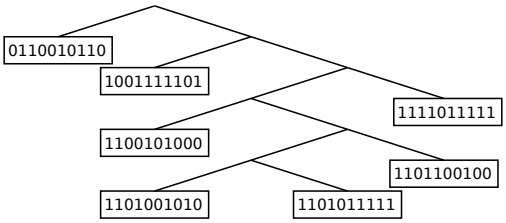
> Hint: Think about how should the shardkv client and server deal with ErrWrongGroup. Should the client change the sequence number if it receives ErrWrongGroup? Should the server update the client state if it returns ErrWrongGroup when executing a Get/Put request?

Describe a sequence of events that demonstrates a problem if the client changes the sequence number when it receives ErrWrongGroup.

Answer: Client sent request, lost reply, resent with new seq number, group moved, got ErrWrongGroup, resent to new group, at which point the operation gets applied again.

VII Kademlia

19. [4 points]: Emily A. Cad wrote a Kademlia client. Ben sent Emily a memory dump of his client, out of which Emily was able to reconstruct Ben's routing table:



Recall that when constructing its routing table, a client splits a k-bucket U into two k-buckets only if searching for the node ID in the tree reaches U. What is Ben's node ID? Choose one answer below:

A. It is 0110010110.

B. It is 1101001010.

C. It is 1101011111.

D. It starts with 11010, and is not B, nor C.

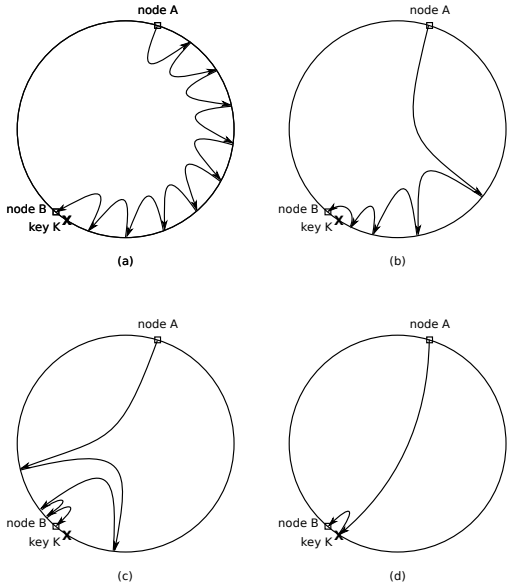
E. Either B or C.

Answer: D. IDs are added to the tree when Ben's client receives messages from Ben's peers. It is those peer addresses that are added to the tree, so Ben's ID should not be in the tree. The structure of the tree is determined by the splits, providing Ben's ID's prefix.

20. [4 points]:

Node A in a Kademlia network is looking up a key K, for which node B is responsible. Which of the following would be most representative of the lookup using Kademlia's XOR metric? Assume $\alpha = k = 1$, and that the nodes have correct routing tables, i.e., in every client, if a node exists in a k-bucket's range, then that k-bucket is not empty.

Answer: C. The routing tables are built such that at every hop, the lookup gets "twice" closer to the queried key, leaving answers B and C. The traversed nodes have more and more prefix bits in common with the queried key, but the actual numeric ID might increase and decrease throughout the run (as in C). Also, node A is in the right half (would have prefix 0*), while the key K and node B are in the left half (with prefix 1*), the first hop should jump to from A to a node on the left half. Answer A has constant-size jumps (should be decreasing in size). Answer D sends a query to a node with ID K, but there cannot be a node there because question states B is responsible for K (so it is the closest node).



VIII Operational systems

21. [4 points]:

Ben Bitdiddle has built a distributed system, and to help him debug problems, he put in many `print` statements throughout the code (often multiple print statements in a single function) that get redirected to a log file. Name three problems that Ben is likely to run into as he scales up his system.

Answer: Logs grow to a large size and require their own infrastructure. Print statements should include request IDs, and be correlated with one another. Print statements might not be reporting enough information. Print statements might be hard to parse in an automated fashion to look for problems.

IX 6.824

We'd like to hear your opinions about 6.824. Any answer, except no answer, will receive full credit.

22. [2 points]: What would you recommend changing for next year?

Answer: More time for final project. More answers in lecture notes (no unanswered questions). Record lectures on video. Avoid race conditions in lab code. More non-storage topics (e.g., Spark). Avoid inter-lab dependencies, or release solutions (x2). Conceptual overview of paper before reading it. Describe background for paper before reading it. More options for final project (x8; be less strict about topic choice). More time for final project (x3). Broader labs (x4; Spark, P2P). No hard deadlines for homework. Late days instead of late hours. Too much Paxos in labs (x2). Clarify expectations for final project (x2). More focus on key concepts, not old papers (x3). Recitations. Ask students to scribe (instead of staff posting lecture notes). Fewer papers and more time on each paper (x2). Better code review interface (x2). Provide quizzes from past years. Make guest lecturers more relevant. More coding, less quizzes (x5). Better lab test cases. Fewer guest lectures. Explaining lab 2 could use more love; it is difficult to understand why it works. Midway check on project. A review of concurrency models. Don't cover same topic too much (Ficus, Bayou, Ivy and TreadMarks). Make the lab tests more mean. In addition to Ficus and Bayou, add Tra. Code reviews and required questions+answers for lecture were almost useless. More focus on Kademlia and Bitcoin. Problem sets instead of quizzes. More weight on final project. More recent papers.

23. [2 points]: Is there one paper out of the ones we have covered in the second half of 6.824 that you think we should definitely remove next year? If not, feel free to say that.

Answer: Bitcoin (covered in 2 other classes). Ficus (x4; replace with Tra). PNUTS (x5; unclear; already in 6.033). AnalogicFS (x4; silly). Bayou (x4; too simple, long). Dynamo (x2; similar to PNUTS). IVY or Treadmarks (x2; too much DSM). Treadmarks. Ficus or Bayou (redundant). Akamai (x7). Kademlia (x2; unclear). Spark. Add Raft. Add something on MPI.

End of Quiz