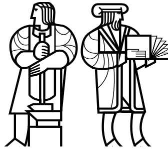


Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.824 Spring 2014

Quiz I Solutions

Histogram of grade distribution

This page intentionally left blank.

I MapReduce

After writing several applications with MapReduce (as described by Dean and Ghemawat), Ben finds it annoying that Map and Reduce jobs cannot share state. He changes the MapReduce API so that the `Map` and `Reduce` functions can invoke key/value operations (i.e., `Get` and `Put`) on a key/value store with the semantics as in the labs. With this API, a map job can perform a `Put` operation on a key that another map job can `Get`.

1. [7 points]: Ben finds that applications that use `Get` and `Put` don't always produce the same result, if he runs the application several times. Give a scenario how this can happen. (You can assume Ben implemented `Get` and `Put` correctly.)

Answer: If a Map job fails, it will re-execute and thus also re-execute any `Get` and `Put` operations again. A re-run of a Map job is likely to read values from the key/value store that are different with the values returned during the first run, and thus results in different outcomes.

This page intentionally left blank.

II Non-determinism and replicated state machines

Ben changes the semantics of `Put` and `PutHash` so that these operations record the last time that each key was updated. He changes `Get` to return the value for a key *and* the last time it was updated.

2. [7 points]: Describe how you would modify lab 3 to implement the modified API correctly: i.e., the key-value servers always return the same, correct time stamp for a `Get` operation, even when clients retry `Put` and `PutHash` operations and servers fail. You can assume that the servers have a function `GetTime` for reading the local clock.

Answer: The server that receives the client request reads the clock and proposes it along with the reqeust to Paxos. Replicas use the time stamp from the Paxos log when applying the operation, instead of reading its own clock.

III Not quite Replicated State Machines

Ben has come up with a simple solution to lab 2B. The pseudocode for his solution is shown on the next page. This pseudocode captures the following strategy:

- The client sends an RPC tagged with its client ID and a unique sequence number to the primary;
- The primary forwards the request to the backup;
- The backup performs the operation (unless it was already performed, in which case it looks up the result from its duplicate detection table), updates its RPC duplicate detection table, and sends back to the primary both the *result* of the operation (possibly from the duplicate detection table) *and* the current (new) value of the corresponding key (not from the duplicate detection table);
- The primary updates its key/value store with the value provided by the backup (instead of re-executing the operation), updates its duplicate-detection table, and sends the response provided by the backup to the client;
- If the backup told the primary it couldn't perform the operation (e.g., because it is in a view change), then the primary tells the client to retry.

Ben points out to Alyssa that his design has only *one* retry loop: namely at the client. In his implementation, there is no retry loop at the primary (or backup); if the primary times out waiting for a response from the backup, it returns an error to the client, and it is the client's job to retry (with the same client ID and sequence number). Furthermore, he argues that his implementation is correct, because it has a well-defined commit point for each operation (e.g., when the backup executes an operation, and updates its key/value table and duplicate-detection table). Alyssa, however, is suspicious about Ben's implementation.

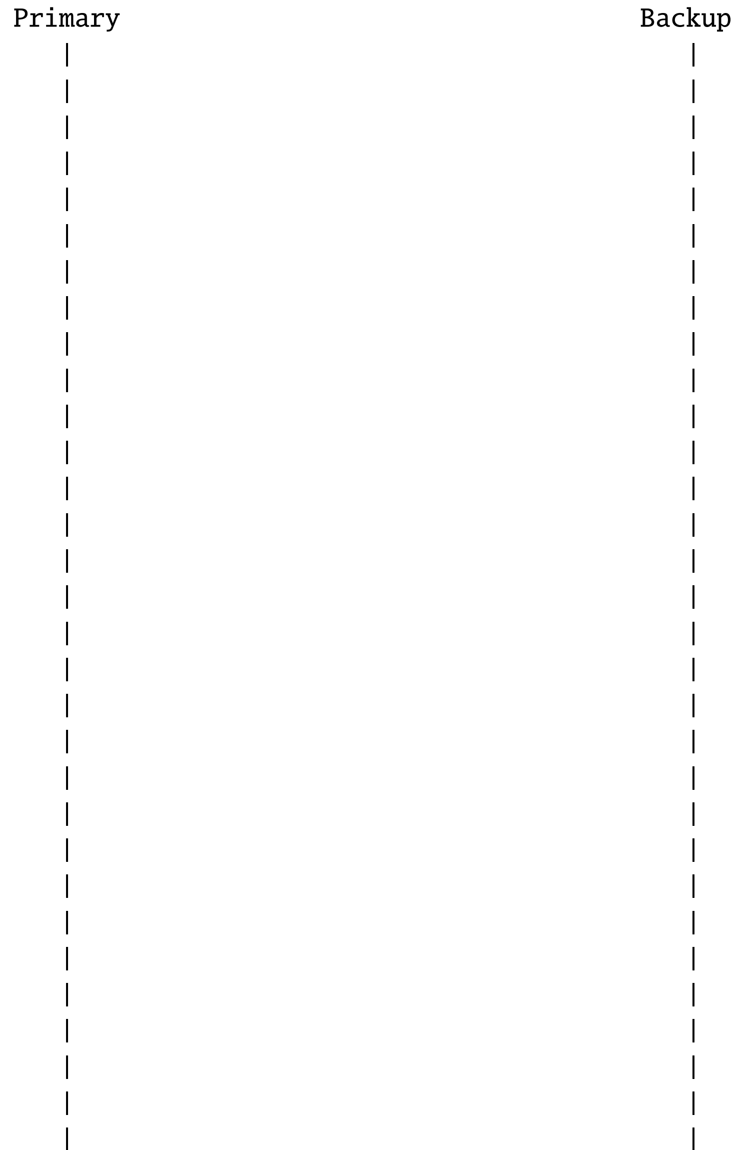
Ben's pseudocode for lab 2B:

```
client_puthash(k, v):
  seq = choose sequence id
  while True:
    send (puthash, client_id, seq, k, v) to primary
    if success: return result
    else ask view server for primary

primary_puthash(client_id, seq, k, v):
  hold global lock
  if (client_id, seq) in dups:
    return dups[(client_id, seq)]
  if there is a backup:
    send (backup_puthash, cur_viewnum, client_id, seq, k, v) to backup
    if error: return error
    get (result, newval) from backup's reply
    db[k] = newval
  else:
    result = db[k]           // result is prev value
    db[k] = hash(db[k] + v)  // do the puthash
  dups[(client_id, seq)] = result
  return result

backup_puthash(arg_viewnum, client_id, seq, k, v):
  hold global lock
  if arg_viewnum != cur_viewnum: return error
  if (client_id, seq) in dups:
    result = dups[(client_id, seq)]
  else:
    result = db[k]           // result is prev value
    db[k] = hash(db[k] + v)  // do the puthash
    dups[(client_id, seq)] = result
  return (result, db[k])
```

Ben's pseudocode for lab 2B, repeated for your convenience (so you don't have to flip pages):

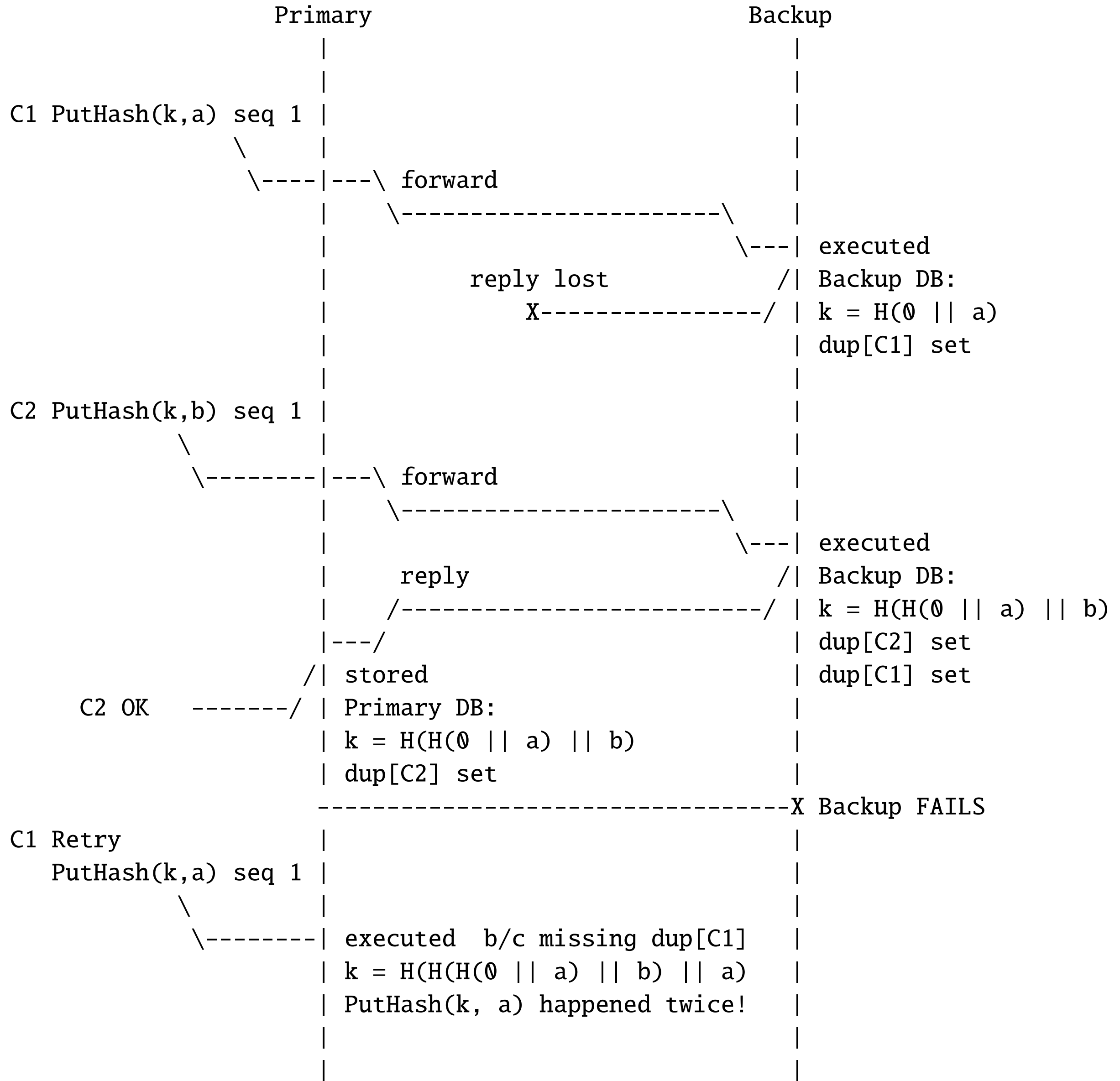
```
client_puthash(k, v):
  seq = choose sequence id
  while True:
    send (puthash, client_id, seq, k, v) to primary
    if success: return result
    else ask view server for primary

primary_puthash(client_id, seq, k, v):
  hold global lock
  if (client_id, seq) in dups:
    return dups[(client_id, seq)]
  if there is a backup:
    send (backup_puthash, cur_viewnum, client_id, seq, k, v) to backup
    if error: return error
    get (result, newval) from backup's reply
    db[k] = newval
  else:
    result = db[k]           // result is prev value
    db[k] = hash(db[k] + v)  // do the puthash
  dups[(client_id, seq)] = result
  return result

backup_puthash(arg_viewnum, client_id, seq, k, v):
  hold global lock
  if arg_viewnum != cur_viewnum: return error
  if (client_id, seq) in dups:
    result = dups[(client_id, seq)]
  else:
    result = db[k]           // result is prev value
    db[k] = hash(db[k] + v)  // do the puthash
    dups[(client_id, seq)] = result
  return (result, db[k])
```

3. [15 points]: In the presence of failures, Ben's implementation doesn't guarantee at-most-once RPC. Show a timing diagram with RPC requests and responses that demonstrates incorrect behavior for `PutHash` operations. For each request sent to the primary, clearly indicate which client sent it, and what the sequence number was. Clearly indicate any messages lost between the primary and the backup, and any primary or backup failures.



Answer:

See the diagram below for a sequence of events that causes a PutHash to be executed twice.


Primary
Backup
C1 PutHash(k,a) seq 1
forward
executed
reply lost
Backup DB:
k = H(0 || a)
dup[C1] set
C2 PutHash(k,b) seq 1
forward
executed
reply
Backup DB:
k = H(H(0 || a) || b)
dup[C2] set
dup[C1] set
stored
C2 OK
Primary DB:
k = H(H(0 || a) || b)
dup[C2] set
Backup FAILS
C1 Retry
PutHash(k,a) seq 1
executed b/c missing dup[C1]
k = H(H(H(0 || a) || b) || a)
PutHash(k, a) happened twice!

This page intentionally left blank.

IV Flat Datacenter Storage

4. [8 points]: In FDS (by Nightingale et al.), a client sends an operation (e.g., `Write`) directly to the replicas that are affected by that operation. For a few operations (e.g., `ExtendBlobSize`), it uses a two-phase commit protocol. Ben replaces the FDS protocol for `ExtendBlobSize` with a plan similar to what FDS uses for `Write`: it sends the `ExtendBlobSize` to all replicas, and when it gets a response from any one of the replicas (the first one to respond), it gives that response back to the application. Describe an application that fails with Ben's design but succeeds on original FDS.

Answer: Several clients try to extend a log file. Each one wants its own part of the log. One client issues the ExtendBlobSize, and gets an ok reply from one replica. The ExtendBlobSize requests to other replicas from this client are lost. The other client issues the ExtendBlobSize concurrently, and gets an ok reply from another replica. Then the two clients will write the same tract, which is wrong and cannot happen in original FDS.

5. [8 points]: Consider the PREPARE handler in a replica, which sends back either OK or ABORT back to the coordinator, for the FDS `ExtendBlobSize` 2PC protocol. In what circumstances must the PREPARE handler return ABORT?

Answer: When that replica is prepared to order another ExtendBlobSize first, but hasn't committed yet and the new PREPARE doesn't have a new TLT version than the prepared one.

V Paxos

Recall the Paxos pseudocode from Lecture 5:

```
    --- Paxos Proposer ---

 1  proposer(v):
 2    choose n > n_p
 3    send prepare(n) to all servers including self
 4    if prepare_ok(n, n_a, v_a) from majority:
 5      v' = v_a with highest n_a; choose own v otherwise
 6      send accept(n, v') to all
 7      if accept_ok(n) from majority:
 8        send decided(v') to all

    --- Paxos Acceptor ---

 9  acceptor state on each node (persistent):
10    n_p      --- highest prepare seen
11    n_a, v_a --- highest accept seen

12  acceptor's prepare(n) handler:
13    if n > n_p
14      n_p = n
15      reply prepare_ok(n, n_a, v_a)


16  acceptor's accept(n, v) handler:
17    if n >= n_p
18      n_p = n
19      n_a = n
20      v_a = v
21      reply accept_ok(n)
```

6. [10 points]:

Ben is trying to minimize the amount of storage that each Paxos instance requires on disk. The pseudocode from lecture requires storing n_p, n_a, and v_a on disk. Ben decides that it may be sufficient to just store n_p and v_a on disk, and if a node reboots, the node sets n_a to the saved n_p value.

Is this modified Paxos protocol correct? Circle one of the two answers, and either explain why it is correct, or give a counter-example timing diagram along the lines shown in the lecture notes from Lecture 5 (use `p1` to designate a Prepare message with $n = 1$, `a2v3` to designate an Accept message with $n = 2$ and $v = 3$, and `reboot` to designate a node restarting). You do not have to include the state after the reboot (though you can draw it on the side for your own benefit).

Yes, it is correct, because:

Answer: No, it is not correct.

No, it is not correct, and here is a timing diagram:

S1:

S2:

S3:

Answer: No, it is not correct. Counter-example:

S1:	p1		p2	a2v2				
S2:	p1	a1v1			p3	reboot	p4	a4v1
S3:	p1		p2	a2v2			p4	a4v1

This page intentionally left blank.

Ben is building a key-value store in the style of lab 3, except that the `Put` and `PutHash` operations do not return any value—that is, they either succeed or fail.

Ben wants to achieve low latency for `Put` and `PutHash` operations, so his server code for both of these operations returns to the client immediately after reaching consensus on that specific operation in some slot in the overall sequence of Paxos instances. The server will execute the operations later on (e.g., when a subsequent `Get` arrives).

For `Get` operations, Ben follows the suggested plan from lab 3: reach consensus on the operation in some Paxos slot, execute all slots up to that one, and return the result to the client.

7. [8 points]: Does Ben's key-value store provide linearizability? A key-value store *provides linearizability* if, whenever operation A completes before B starts, then no client can observe the effects of B without also observing the effects of A.

Circle one of the following choices, and explain why or why not.

Yes, Ben's key-value store provides linearizability, because:

No, Ben's key-value store does not provide linearizability, because:

Answer: No. A client can issue a `PutHash` to one replica, get its operation committed in some position in the log, then talk to another server to commit a subsequent `PutHash` operation in an earlier hole in the log. Since the first `PutHash` does not wait to fill in all previous holes, this leads to the two `PutHash` operations executing in an order inconsistent with the causal order observed by the client.

This page intentionally left blank.

8. [8 points]:

Ben decides to switch his lab 3 key-value server to EPaxos instead of regular Paxos with a sequential log, and tries to implement the optimization from the previous question again: for `Put` and `PutHash` operations, the server returns after committing the operation to EPaxos, but without executing the operation. Does EPaxos provide linearizability in this case? (The definition of linearizability is the same as in the previous question.)

Yes, Ben's key-value store in EPaxos provides linearizability.

No, Ben's key-value store in EPaxos does not provide linearizability.

Answer: Yes, Ben gets linearizability. EPaxos orders conflicting operations on the same key by their seq#. If one operation commits, and another conflicting operation is issued after that, it is guaranteed to get a higher seq# and will be executed later.

This page intentionally left blank.

VI Spanner

9. [8 points]:

Suppose that client A performs a `Put` in Spanner, sends a message to client B, and B does a `Get` on the same key. How does Spanner ensure that the commit time of `Get` is higher than the commit time of the `Put` (that is, ensuring that the `Get` will see the `Put`'s value)?

Answer: Spanner will assign the Put a commit timestamp within [TT.now().earliest, TT.now().latest]. Due to commit wait, Spanner will make sure when Put is completed, true time is after the commit timestamp of the Put. For Get, Spanner will assign it either the commit timestamp of the Put or TT.now().latest (see 4.2.2). Spanner will then only execute the Get until it sees all writes that may happen before the commit time of the Get, which means Get will see the Put.

This page intentionally left blank.

VII Harp

10. [15 points]:

Suppose Harp is running in a 5-server configuration, with servers S1, S2, S3, S4, and S5. S1 is the designated primary, S2 and S3 are designated backups, and S4 and S5 are designated witnesses. For each of the following scenarios, say whether Harp can continue running (True) or not (False). By crash, we mean that the machine's CPU immediately stops executing, and the memory contents are lost.

(Circle True or False for each choice.)

A. True / False S1 and S2 crash. **Answer:** True.

B. True / False S1 and S2 and S3 crash, reboot, and come back up. **Answer:** False, could have lost operations.

C. True / False S1 and S2 and S5 crash, reboot, and come back up. **Answer:** True, S3 has a copy of all committed ops.

D. True / False S3 and S4 and S5 crash, reboot, and come back up. **Answer:** True, S1 and S2 have copies of all committed ops.

E. True / False S3 and S4 and S5 crash and lose their disk, reboot, and come back up with clean disks. **Answer:** False, the crashed servers may have formed a new view, processed operations, and lost them.

VIII 6.824

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

11. [2 points]: What aspects of the labs were most time-consuming? How can we make them less tedious?

Answer: debugging, race conditions, deadlock (many); suggestion to add debug messages to RPC library (print messages and result); explain test cases (many); provide a correct paxos after 3a; fix labs to not trigger go's race detector (kill); learning go / need more library function hints; don't know what code would be reused between labs (refactoring); not being able to suppress output made debugging hard (x2); some suggestions were mandatory (piggyback Done on RPC messages), be upfront about this; code review; duplicate detection; more detailed specs (many);

12. [2 points]: Are there other things you'd like to see improved in the second half of the semester?

Answer: due dates on sunday; later deadline for paper questions/answers; less papers, more labs; better explanation of papers (x2); code review (x2); fewer readings / quizzes (x2); post answers to lecture questions (x3); make lecture question answers public after 11pm; more case studies (e.g., dropbox; x2); less coding; later deadline for questions; release lab grades and reviews sooner (x4); better lecture notes WITH GRAPHS (x2); allow single paper to span multiple lectures; more emphasis on final project and less on labs; more practical systems, fewer old papers; no papers, MORE CONCEPTS; focus more on high level concepts (2PC, 2PL, x2); update website with announcements; should have spent more time on spanner (actual lecture on concepts); feedback on paper question answers; be more clear what will be on quiz; office hours on different days/times (suggested Th,F,Sat); move to room with wider seats; better code review site (suggest using 6.005 caesar system for code review); more review sessions for next quiz; focus more on performance, not reliability; more labs; 2 tests/labs/projects are too much work;

13. [2 points]: Is there one paper out of the ones we have covered so far in 6.824 that you think we should definitely remove next year? If not, feel free to say that.

Answer: mapreduce (duplicate with 033, x2); pnuts (duplicate with 033); lynx (many), b/c not used in practice / redundant compared to Spanner spanner (x4); FDS (confusing, not well explained, x3); hypervisor (old, many); epaxos (too hard/confusing, redundant, x5); harp (many); treadmarks (old, x2); paxos paper (still confusing, does better one exist?); none (many); spanner later (x2); suggest Raft;

End of Quiz