Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.824 Distributed System Engineering: Spring 2013

Quiz I Solutions

I Paxos

Here is the Paxos pseudo-code:

```
proposer(v):
  while not decided:
    choose n, unique and higher than any n seen so far
    send prepare(n) to all servers including self
    if prepare_ok(n_a, v_a) from majority:
      v' = v_a with highest n_a; choose own v otherwise
      send accept(n, v') to all
      if accept_ok(n) from majority:
        send decided(v') to all

acceptor's state:
  n_p (highest prepare seen)
  n_a, v_a (highest accept seen)

acceptor's prepare(n) handler:
  if n > n_p
    n_p = n
    reply prepare_ok(n_a, v_a)
  else
    reply prepare_reject

acceptor's accept(n, v) handler:
  if n >= n_p
    n_p = n
    n_a = n
    v_a = v
    reply accept_ok(n)
  else
    reply accept_reject
```

1. [5 points]: Explain why the "decide" phase of the Paxos protocol is not necessary: explain why prepare and accept are sufficient for peers to discover the agreed-on value.

Answer: Any node that needs to find out the agreed value (if any) can start acting as a proposer. If a value has already been agreed on, it will end up reaching agreement again on that value.

2. [10 points]: Suppose that all Paxos peers are alive and reachable and correctly executing the pseudo-code, and that the network delivers all messages with low delay. Describe a scenario in which two Paxos proposers start at the same time, but neither manages to reach agreement after its first round of prepare and accept messages. Describe the messages sent in your scenario, and the relevant details about the order in which the messages are sent and/or received.

Answer: Proposer P1 sends out prepares with n=1, and gets a full set of successful replies. Proposer P2 sends out prepares with n=2 and gets a full set of successful replies. P1 sends out its accepts, with n=1, and they are all rejected. P1 immediately starts the next round with n=3 and sends out prepares. Then P2's accepts will be rejected. It would be better for P1 to sleep for a while before starting the next round.

II FDS

3. [10 points]: Section 3.2 of the Flat Datacenter Storage (FDS) paper explains how the system copes with the failure of a single tractserver. The metadata server deletes the tractserver from each TLT row in which it appears, and replaces each such instance with a randomly chosen live server. Suppose, instead, that the metadata server maintained a pool of unused servers, and used one of these servers to replace the failed tractserver. How would this affect total recovery time? Assume that the FDS system is idle except for the work required to recover.

Answer: It will increase recovery time. FDS needs to create a new copy of all the data stored on the failed server. Assuming two servers per TLT row, part of that data is stored on each server which shared a TLT row with the failed server. With the random scheme, each of those servers sends its fraction of the data to a (usually) different randomly selected server, so no server reads or writes more than a fraction of the data. With the suggested idle server scheme, all the reconstructed data has to be written to the disk on that one new server; disk throughput is likely the limiting resource, so the recovery will take much longer.

III Spanner

Fred and Sally are using the Spanner storage system. They wonder how Spanner would behave with the code from Lecture 9 that we used to help decide if a storage system is sequentially consistent:

```
x = 1              y = 1
if y == 0:         if x == 0:
  print yes          print yes
```

Fred and Sally translate this into Spanner client code as follows:

```
// Fred's client:
start a read/write transaction
  Write(key="x", value="1")
end transaction
start a snapshot read
  print Read(key="y")
end

// Sally's client:
start a read/write transaction
  Write(key="y", value="1")
end transaction
start a snapshot read
  print Read(key="x")
end
```

Keys "x" and "y" both start with value "0". The two clients start executing their code at about the same time. Spanner executes all the operations successfully, and both clients get replies to all their Spanner requests. Spanner assigns the two read/write transactions (and the two Write()s) different timestamps.

For each of the following situations, indicate whether both clients **could** print "0". For each, briefly explain your answer.

4. [5 points]: Each client issues its snapshot read with client-provided timestamp equal to TrueTime.Now().earliest at that client.

Answer: Yes. TrueTime doesn't guarantee anything about Now().earliest other than it is before the absolute time. Thus both "earliest" values might be before the timestamp Spanner assigns to either write, so both clients might read zero.

5. [5 points]: Each client issues its snapshot read with client-provided timestamp equal to TrueTime.Now().latest at that client.

Answer: No. Spanner's commit delay mechanism ensures that a write does not complete until the absolute time has passed. TrueTime guarantees that Now().latest is after the absolute time. So each client reads at a snapshot time after its write's timestamp. Suppose Fred's client reads y=0. That means Fred's client's Now().latest is less than Sally's write's timestamp; and also that Fred's write's timestamp is less than Sally's write's timestamp. That, finally, means Fred's write's timstamp is less than the snapshot time at which Sally will read, so Sally's read is guaranteed to see x=1.

6. [5 points]: Each client issues its snapshot read with client-provided timestamp equal to the timestamp that Spanner assigned to that client's Write.

Answer: No. The explanation is essentially the same as the previous question: if Fred reads y=0, his read and write must have timestamps before Sally's write, and thus before Sally's read, so Sally must read x=1.

IV Distributed Shared Memory

Suppose we had a version of the system described in Li and Hudak's Memory Coherence in Shared Virtual Memory Systems. Our system, called Byte-IVY, operates on pages that are just one byte long. This is possible because the computer we're using has a hardware page size of just one byte: there is a distinct page table entry for each virtual address. Byte-IVY, just like IVY, has a manager for each page, but of course there are now many more pages.

You should assume that Byte-IVY has no bugs. You can base your answer on any of the versions presented in the paper; which you choose should not affect your answer.

> **7. [10 points]:** Would Byte-IVY benefit from the write diffs technique described in the TreadMarks paper? Why or why not? If your answer is "yes", describe a specific example in which Byte-IVY would benefit from write diffs.

Answer: No. With byte-sized pages, Byte-IVY eliminates the problem that a single-byte write causes an entire (larger) page of mostly-unmodified data to be sent over the network. So write diffs would not save any communication cost. Though if you assume bit-level write diffs, the answer is "yes."

> **8. [10 points]:** Would Byte-IVY benefit from TreadMarks' Lazy Release Consistency? Why or or why not? If your answer is "yes", describe a specific example in which Byte-IVY would benefit from LRC.

Answer: No. Byte-IVY only sends modified data to machines that read that data, since IVY page transfers are driven by read faults. Thus, for properly locked code, Byte-IVY would send the minimum amount of data possible; there's no room for LRC to reduce that. Perhaps LRC could help by eliminating IVY's invalidate messages.

V CBCAST

Three machines (M1, M2, and M3) use CBCAST to order messages, as described in Lecture 11. Recall that a CBCAST sender sends to all receivers. The CBCAST library at each receiver buffers incoming messages, and only delivers them to the application when it is correct to do so under causal consistency. The network never drops a packet.

The application running on M1 uses CBCAST to send a message "X". CBCAST on M2 receives the message and delivers it to the application. After the application on M2 sees "X", it uses CBCAST to send "Y". A little later the application on M1 sends message "Z". All messages are eventually delivered to the the applications on M1, M2, and M3.

9. [5 points]: CBCAST sends a vector clock value with each message. Write down a vector clock value for each message that's consistent with the above scenario.

```
X:  <1, 0, 0>
Y:  <1, 1, 0>
Z:  <2, 0, 0> or <2, 1, 0>
```

10. [5 points]: The CBCAST library on M3 delivers the messages in some order to the application on M3. Which of the following orders is possible? Circle the orders that are possible; draw a line through the orders that are not possible.

```
X Y Z -- possible
X Z Y -- possible
Y X Z -- impossible
Y Z X -- impossible
Z X Y -- impossible
Z Y X -- impossible
```

VI Labs

Recall that Lab 2B (pbservice) involves a view server, a primary, and (when possible) a backup. Suppose there's a view in which server S0 is the primary and S1 is the backup. S1 obtained a copy of S0's complete state at the start of the view, and for a while S0 and S1 have been happily processing client requests.

At some point S0 tries to forward a client request to S1, but S0's RPC library returns an error indicating it received no reply from S1. S0 has not heard of any view change from the view server.

> **11. [5 points]:** Would it be OK for S0 to execute the client request and return a success indication to the client? Why or why not?

Answer: No. That would result in the primary and backup no longer being replicas. If S0 were later to crash, and S1 took over, it would have different data. Clients reading data at the time of the crash would see values change without any client having issued a Put.

> **12. [5 points]:** Would it be OK for S0 to **not** execute the client request, and to return a failure indication to the client? Why or why not?

Answer: No. The network might have dropped the RPC reply after S1 executed the request. Again this would cause the two servers no longer to be replicas.

For Lab 3B (kvpaxos), your 6.824 class-mate Robert doesn't understand why the servers need to detect duplicate requests sent by the code in `client.go`. After all, he says, a duplicate Get is obviously harmless, and a duplicate Put just writes the same value twice, again clearly harmless.

> **13. [10 points]:** Explain why duplicate client request detection is necessary in Lab 3B. Describe how a duplicate client request could occur, and present a specific scenario in which it would be harmful if not detected.

Answer: Client C1 might send Put(x,1) to server S1. S1 is having network problems so it cannot immediately execute the Put. Meanwhile C1's Clerk times out and re-sends the Put to a different server, which does execute it. Client C2 does a Get(x) and sees the value 1; then C2 does a Put(x,2); then does Get(x) again and sees value 2. Finally S1 executes the original Put(x,1). The client application only made two calls to Clerk.Put(), but clients reading from the system see three changes to the value for x.

VII 6.824

14. [5 points]: The labs are entirely new this year. Did you find them useful? How could we improve them?

Answer: Better documented test code. Better debugging support. More labs, on a wider array of topics.

15. [5 points]: What non-lab aspects of 6.824 would you most like to see changed?

Answer: Post answers to the paper questions. Include answers to questions raised in lecture notes. Faster grading of labs. Reading guides for papers. Easier-to-read papers. Fewer exams.

End of Quiz I Solutions