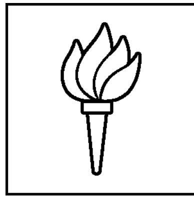

Computer Science Department

New York University

G22.3033-001 Distributed Systems: Fall 2015

Quiz I

In order to receive credit you must answer the question *as precisely as possible*. You have 80 minutes to answer 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, be sure to write down any assumptions you make. Be neat. If we can't understand your answer, we can't give you credit!

THIS IS AN OPEN BOOK, OPEN NOTES QUIZ.

I (xx/25)	II (xx/20)	III (xx/20)	IV (xx/20)	Total (xx/85)

Name:

NYU ID:

I Multiple choice questions (25 points):

A. Which of the following statements are true about MapReduce?

1. In a MapReduce system, users can write non-deterministic Mapper and Reducer functions and get expected results.
2. If machines have the same hardware configuration, then all Reducers take the same amount of time to finish.
3. MapReduce cannot start executing any user-defined Reduce function until all Mappers have finished.
4. MapReduce may incur duplication execution of a Mapper or Reduce task.

B. Which of the following statements are true about RPC?

1. It is difficult for users to program with an RPC system that provides exactly-once-execution guarantee.
2. If an RPC with at-most-once-guarantee returns an error, then the request has not been executed at the server.
3. If an RPC with at-most-once-guarantee returns successfully, then the request has been executed at the server.
4. In a key-value store, it is safe for the server to process a duplicate GET request.
5. In a key-value store, it is safe for the server to process a duplicate PUT request.

C. Which of the following things are true about a linearizable primary/backup system?

1. The primary handles all PUTs issued by clients.
2. Either the primary or the backup can handle GETs issued by clients.
3. The primary must synchronously replicate the PUT request to backups before replying to the client.
4. The primary can only handle one PUT or GET operation at a time.

D. Which of the following things are true about linearizability?

1. Linearizability is a replication protocol.
2. A linearizable system must be fault tolerant.
3. One can build a linearizable storage system using Paxos or Primary/backup.
4. Linearizability provides a stronger guarantee than sequential consistency.

E. Which of the following things are true when running Paxos?

1. When running with 3 nodes, Paxos can achieve consensus in the face of a malicious node that does not follow the protocol.
2. There is no guarantee that Paxos terminates with a consensus value within a threshold amount of time.
3. The accepted value of a node can never change.
4. Paxos allows multiple proposers to propose concurrently.

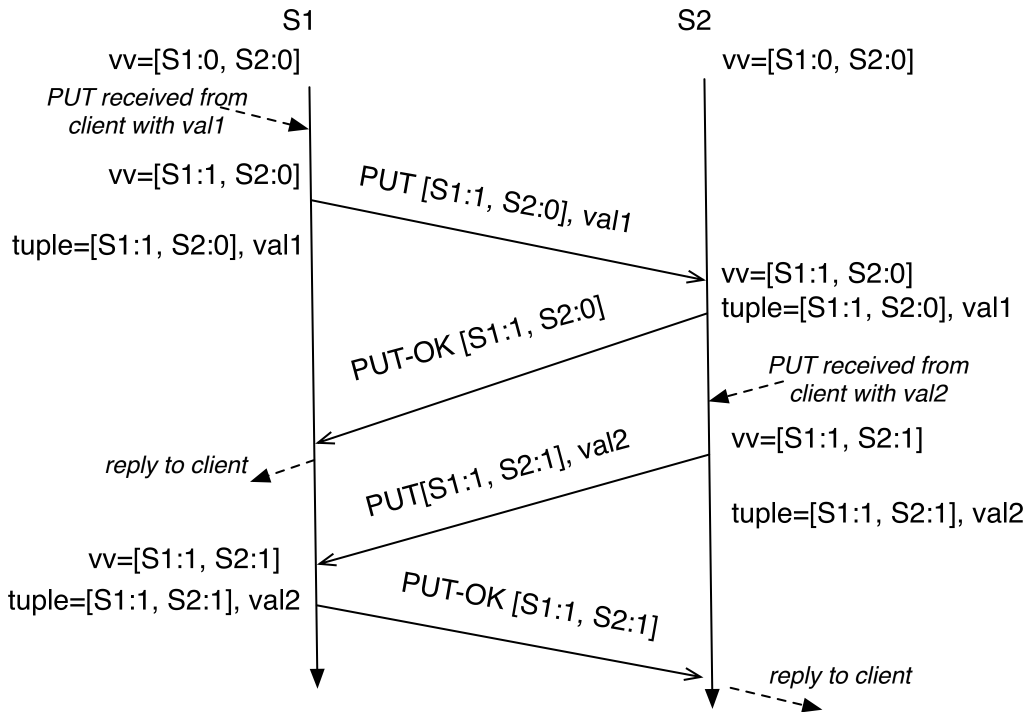

Figure 1: Server S_1 processes a client request. Then server S_2 processes another client request.

II Consistency in Key-value stores (20 points)

Ben Biddle got inspired by Amazon's Dynamo system and decides to build a small-scale replicated key-value stored based on the idea of version vectors.

Ben's key-value store has two servers (S_1 and S_2) and many clients. For any given GET or PUT, a client is free to ask *either* of the two servers to process the request.

Each server maintains its version counter and also keeps track of the largest known version number of the other server. This is referred to as a server's version vector. All messages contain the sender's version vector which is used by the receiver to update its version vector. Suppose the message contains [S_1:i, S_2:j] and the receiver S_1's local version vector is [S_1:i', S_2:j']. The, S_1's new version vector is [S_1:max(i,i'), S_2:max(j,j')]

Suppose a client sends server S_1 a PUT. To process this request, first, S_1 increments the counter associated with S_1 in its version vector and assigns the vector to PUT. Then, S_1 sends the versioned PUT request to S_2 and also processes it locally. To process a versioned PUT request, a server simply stores the versioned tuple locally, overwriting the existing tuple if one exists. The server replies to the client after the versioned PUT has been processed by both servers.

Below, Ben illustrates how his protocol works in a scenario where S_1 handles a PUT and then S_2 handles another PUT.

1. [5 points]: Alyssa Hacker is trying to convince Ben that his design of simply overwriting the existing tuple does not provide eventual consistency. Please help Alyssa give a concrete example in which the tuple stored by S_1 can become different from that stored at S_2 permanently. Please structure your example in a graphical format like that shown in Figure 1

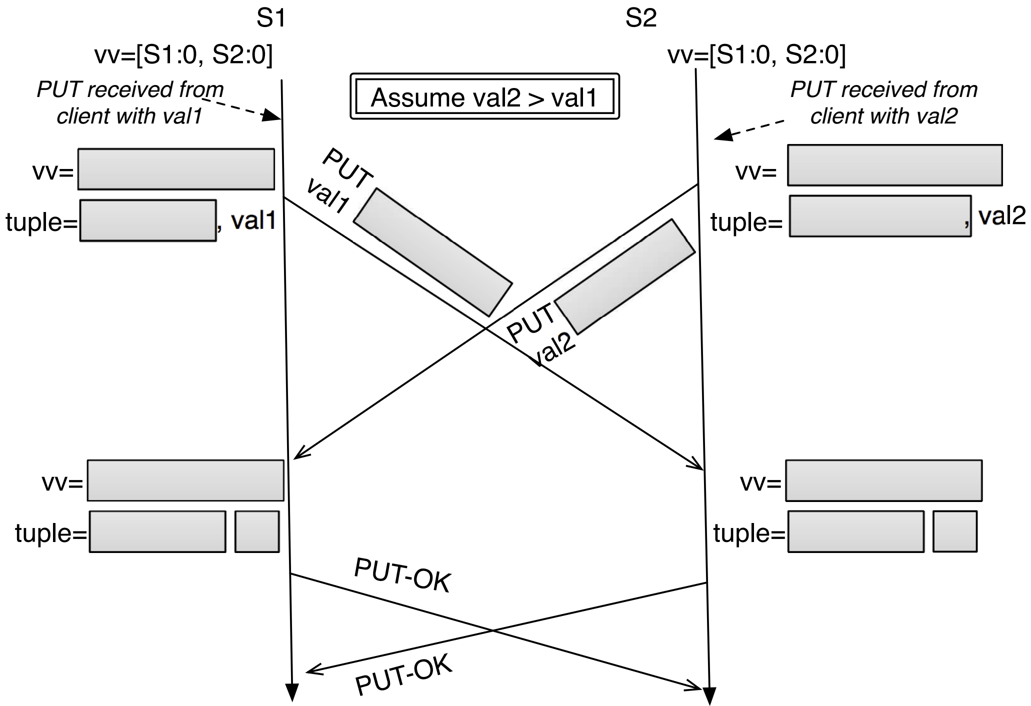

Figure 2: Servers S_1 and S_2 process two PUTs concurrently. Fill in the grey boxes according to Ben's new design, assuming $val2 > val1$.

2. [5 points]: To address Alyssa's concern, Ben has come up with a new design. When handling a versioned PUT request, a server over-writes its existing tuple if the version vector in the request is larger than that of the existing tuple. If the two version vectors are in-comparable, the server performs conflict resolution: if the tuple in the PUT request contains a larger value, it stores the value in the PUT. Otherwise, it keeps the existing tuple's value. The version vector stored with the tuple is the merged vector. For two vectors [S_1:i, S_2:j] and [S_1:i', S_2:j'], the merged vector is [S_1:max(i,i'), S_2:max(j,j')]. Please fill in blanks in Figure 2 below illustrating how Alyssa's scheme works with the same example shown earlier.

3. [5 points]: In Ben's design, a server handles a GET request by simplying returning to the client the tuple stored locally. Alyssa points out that doing so violates linearizability. Please illustrate a concrete example showing the violation of linearizability.

4. [5 points]: Can you give a design for how to handle GETs so that linearizability is guaranteed?

III Paxos (20 pts)

When doing Lab 3, Ben Bitdiddle thinks he can speed up Paxos from a three-round to a two-round protocol. In his new protocol, Ben includes the value a proposer would like to use for the concensus in its prepare messages and acceptors accept values when handling the prepare messages (there are no more accept messages). As soon as the proposer receives a majority of prepare-ok messages, it can decide on the consensus value.

```
//v is the value this proposer would like to use for the concensus:
proposer(v):
  choose n, unique and higher than any n seen so far
  send prepare(n, v) to all servers including self
    if prepare_ok(n) from majority:
       send decided(n,v) to all

acceptor's prepare(n, v) RPC handler:
//n is the proposal number
//v is the value
if (n >= n_a) {
   n_a = n
   v_a = v
   reply prepare_ok(n)
} else {
   reply prepare_rejected(n)
}
```

5. [10 points]: Is Ben's modification correct? If not, please explain using a counterexample. (If you are giving a counterexample, please keep it simple by at most 3 total nodes.)

6. [5 points]: Explain (the original, unmodified) Paxos' behavior when the underlying network partitions the set of nodes. Suppose in the current view, the nodes particpating in Paxos are $\{S_1$, S_2, $S_3\}$ and the network partitions the nodes into two groups, $\{S_1, S_2\}$ and $\{S_3\}$. In one group, S_1 and S_2 can communication with each other. In the other group, S_3 can communication with neither S_1 nor S_2.

7. [5 points]: After the network partitions (one partition is $\{S_1, S_2\}$ and the other partition is $\{S_3\}$), a new node S_4 joins the partition containing S_3. Neither S_4 nor S_3 can communication with S_1 or S_2. Can nodes S_4 and S_3 create a new view consisting of $\{S_3$, $S_4\}$? Can nodes S_1 and S_2 create a new view consisting of S_1, S_2?

IV Paxos and Linearizability (20 pts)

Consider the following histories of execution for a key-value store (assuming all key-value pairs have an initial value of 0). P_{req} refers to a `put` request and P_{ok} refers to a reply for the corresponding `put` request. Similarly, G_{req} refers to a `get` request and G_{ok} refers to a reply for the corresponding `get` request. We use superscript $c1$, $c2$, $c3$ to refer to requests issued by clients $c1$, $c2$, or $c3$, respectively.

A. $P^{c1}_{req}(x=1)$, $P^{c2}_{req}(x=2)$, $P^{c1}_{ok}(x=1)$, $G^{c1}(x)$, $P^{c2}_{ok}(x=2)$, $G^{c1}_{ok}(x=0)$

B. $P^{c1}_{req}(x=1)$, $P^{c2}_{req}(x=2)$, $P^{c1}_{ok}(x=1)$, $G^{c3}(x)$, $P^{c2}_{ok}(x=2)$, $G^{c3}_{ok}(x=0)$

C. $P^{c1}_{req}(x=1)$, $P^{c2}_{req}(x=2)$, $P^{c1}_{ok}(x=1)$, $P^{c2}_{ok}(x=2)$, $G^{c3}(x)$, $G^{c3}_{ok}(x=1)$

D. $P^{c1}_{req}(x=1)$, $P^{c2}_{req}(x=2)$, $P^{c1}_{ok}(x=1)$, $G^{c3}(x)$, $P^{c2}_{ok}(x=2)$, $G^{c3}_{ok}(x=2)$

8. [10 points]: Which of the above histories are **not** linearizable? Please explain why.

9. [5 points]:. Ben Bitdiddle decides to design an optimized Lab3 by using less messages for GETs. In Ben's design, each client chooses one server to issue all its requests (i.e. a client can *not* switch among different servers when issuing its requests.). The servers still use Paxos to agree on a log of PUTs operations where log entry i contains the already decided consensus value of Paxos instance i. Ben's key-value store does not support APPEND. To process GETs, a server simply reads the entry with the largest seqno from its local log and return it to the client.

Is Ben's new design linearizable? If not, which of the above histories are possible? Please explain your answer with a concrete sequence of events.

10. [5 points]:. Ben has come up with an alternative plan to optimize GETs. Like before, the servers still use Paxos to agree on a log of PUTs. However, to process GETs, a server S reads the entry with the largest seqno from its local log and also sends a message to another node S'. S' returns its key-value entry with the largest seqno from its own local log to server S. The server S returns the entry with the larger seqno among the two key-value pair (from S and S').

Is Ben's design linearizable (assuming there are exactly three replica servers)? If not, which of the above histories are possible? Please explain your answer with a concrete sequence of events.

Solutions:

Problem I.
A. 3,4
B. 3,4
C. 1,3
D 3,4
E 2,4

Problem II
1. Client 1 issues a PUT request with val1 to S1 and client 2 concurrently issues a PUT request with val2 to S2. S1 assigns version vector <1:0> to val1 and sends it to S2. S2 assigns version vector <0:1> to val2 and sends it to S1.
S2 receives S1's PUT and writes val1. S1 receives S2's PUT and writes val2.

2. The example is the same as described in 1, except when S1 receives S2's PUT, S1 performs a conflict resolution such that its final stored value is val2 with version vector <1:1>. When S2 receives S1's PUT, it also performs a conflict resolution and keeps val2 (instead of overwriting it with val1) with version vector <1:1>.

3. Here's a counterexample. Suppose two PUTs concurrently store val1 and val2 as described earlier. While these two PUTs are going on, client1 obtains GET=val1 from S1. Afterwards, client2 obtain GET=val2 from S2. Afterwards, client2 obtains GET=val1 from S1. No lienarizable system would permit such a sequence of events.

4. One feasible fix is to have each client issue its GET request to both S1 and S2. After obtaining the results from both server, the client returns the key-value pair with the higher version vector if both version vectors are comparable. If the vectors are incomparable, the client performs a conflict resolution and returns key-value pair with the greater value.

Problem III

5. The modification is incorrect.
```
S1:p1 a1vA dA
S2:   a1vA  a2vB
S3:    p2  a2vB dB
```

6. The majority group (consisting S1 and S2) can proceed as usually (to propose and decide any value). The minority group consisting of only S3 cannot decide on any values.

7. No. Nodes S3, and S4 cannot form a new view without being able to get a majority agreeement from the current view (which consists of nodes S1, S2 and S3).

Yes. Nodes S1 and S2 can create a new view consisting of only S1 and S2, since they form a majority of the current view (which consists of nodes S1, S2 and S3).

Problem IV
8.
A. G is issued after P^{c1}{ok}(x=1), so it cannot return x=0 according to linearizability
B. Same as above.

9. No. B is possible. Suppose there are three servers, s1,s2,s3. client c1 sends its PUT request to s1 who proposes and decides the next log entry to be x=1. It relies PUT-ok to c1 and sends the decide message to s2,s3. Before the decide-message from s1 arrives, client c3 sends GET request to s2 and s2 relies with x=0.

10. No. B is still possible. Suppose there are three servers, s1,s2,s3. client c1 sends its PUT request to s1 who proposes and decides the next log entry to be x=1. It relies PUT-ok to c1 and sends the decide message to s2,s3. Before the decide-message from s1 arrives, client c3 sends GET request to s2 who thinks x=0; s2 also asks s3 who also thinks x=0 (since s1's decide message has not arrived at s3 yet.). Thus, s2 relies to the client with x=0.