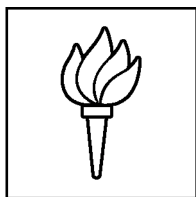

Computer Science Department

New York University

G22.3033-001 Distributed Systems: Fall 2014

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/30)	III (xx/25)	IV (xx/20)	Total (xx/100)

Name:

NYU ID:

I Multiple choice questions (25 points):

Answer the following multiple-choice questions. Circle *all* answers that apply. Each problem is worth 5 points. Each missing or wrong answer costs -3 point.

A. Suppose you are using the Lab 1's original RPC system (i.e. before you add the at-most-once guarantee). Consider the following client code snippet (assume all return values are OK):

```
...
cl->call(kv_protocol::put, "k1", "1", ...);
cl->call(kv_protocol::put, "k1", "2", ...);
cl->call(kv_protocol::put, "k1", "3", ...);
cl->call(kv_protocol::get, "k1", val);
cout << val << endl;
```

What might be the potential resulting output? Let's assume the initial value of key "k1" is "0".

1. 0
2. 1
3. 2
4. 3

B. Which of the following statements are true for an RPC system with at-most-once guarantee?

1. When an RPC call succeeds, the client knows that the server has executed the corresponding handler exactly once.
2. When an RPC call fails, the client knows that the server has not executed the corresponding handler.
3. The server must **not** forget the id and result of every RPC that it ever has processed.
4. If a server restarts after a crash, it must use a new server nonce (i.e. `rpcs::nonce_`) that is different than the one it has used before the crash.

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

1. A linearizable system is also causally-consistent.
2. A linearizable key-value system cannot scale out to use many machines.
3. If a client C1 issues request `PUT(x=1)` before another client C2 issues request `PUT(y=2)`, then all other clients who have seen C2's write (i.e. `GET(y)=2`) cannot observe x's old value prior to C1's write.
4. It is the job of application programmers to ensure that a system is linearizable.

D. Which of the following things are true about causal consistency?

1. Causal consistency can not be realized in a scalable fashion.
2. Causal consistency offers weaker semantics than linearizability.
3. In a causally consistent storage system, data replication across nodes can be done in the background asynchronously.
4. In a causally consistent storage system, if a client C1's write completes before another client C2's read, C2 is guaranteed to see C1's write.

E. Which of the following things are true about transactions?

1. A database must execute transactions serially one after another in order to achieve serializability.
2. A transaction that only reads from the database does not need any concurrency control to ensure serializability.
3. Two phase locking is a concurrency control mechanism to achieve serializability.
4. Under serializability, a transaction T_1 may read the writes of a not-yet-committed transaction T_2.

II Transactions (30 pts)

On SuperAuction.com website, users can sell their items and bid on others' items. The website allows two types of operations, `BidItem` and `EndAuction`. `BidItem` is invoked whenever a user places a bid on an item. `EndAuction` is invoked by a system timer to mark the item as sold. The pseudocode for the two functions is provided below. For simplicity, we assume there's only a single item on sale. Shared variable `highest_bid` stores the item's current highest bid price. Shared variable `sold` stores the sale status of the item.

```
//invoked when a user bids on the item with bidding price bid
BidItem(bid_price):
   if (!sold && bid_price > highest_bid) {
      highest_bid = bid_price
      cout << "success, highest=" << bid_price << endl
   } else if (sold) {
      cout << "failure, item already sold" << endl
   } else {
      cout << "failure, highest=" << highest_bid << endl
   }

EndAuction():
   sold = true;
   cout << "item sold for " << highest_bid << endl
```

For each of the questions below, we assume these initial values: `highest_bid=100`, `sold=false`.

1. **[10 points]:** If `BidItem` is enclosed in a **serializable** transaction, what are the potential printed results when two operations, `BidItem(120)` and `BidItem(200)`, execute concurrently? What are the potential values for shared variable `highest_bid` after both operations finish? Please list all potential outcomes. (Note that we assume `EndAuction` is never invoked here.)

2. [10 points]: If `BidItem` is **not** enclosed in a transaction, what are the potential printed results when two operations, `BidItem(120)` and `BidItem(200)`, execute concurrently? What are the potential values for shared variable `highest_bid` after both operations finish? Please list all potential outcomes. (Note that we assume `EndAuction` is never invoked here.)

3. [10 points]: If `BidItem` and `EndAuction` are each enclosed in a transaction guaranteeing **snapshot isolation**, what are the potential printed results when two operations, `BidItem(110)` and `EndAuction()` execute concurrently? What are the potential values for shared variables `highest_bid` and `sold` after both operations finish? Please list all potential outcomes.

III Paxos (25 pts)

When doing Lab 2, Ben Bitdiddle thinks he can speed up Paxos from a three-round to a two-round protocol. In his proposition, as soon as a node has accepted a value, it returns the value as the final consensus value. Below is the pseudocode of Ben's accept RPC handler function:

```
acceptor's accept(n, v) RPC handler:
//n is the proposal number
//v is the proposal's value
if (n >= n_h) {
   n_h = n
   n_a = n
   v_a = v
   commit(v_a) //take v_a as the consensus value
   reply accept_ok(n)
} else {
   reply accept_rejected(n, n_h)
}
```

4. [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 using ≤ 3 nodes in total and ≤ 2 concurrent proposers.)

5. [5 points]: Alyssa P. Hacker notices that Ben's accept handler does not durably log any data to a node's local disk. She thinks that is not correct if a crashed node comes back online. What are those variables that **must** be logged? Choose all that apply.

1. `n_h`
2. `n_a`
3. `v_a`

6. [10 points]: Ben Bitdiddle argues with Alyssa that his Lab2 implementation does not need to log those values if a crashed node does not join the system with its old node identifier upon recovery. Instead, a recovered node joins using a new unique node identifier. Is Ben correct? If not, please give a counter example. If yes, please give one reason why one might not want to adopt Ben's approach in practice.

IV Linearizability (20 pts)

Consider the following histories of execution for a key-value store (assuming all keys have an initial value of 0). P_{req} refers to a put request while P_{ok} refers to a reply for the corresponding put request. Similarly, G_{req} refers to a get request while G_{ok} refers to a reply for the corresponding get request.

1. $P_{req}(x = 1)$, $G_{req}(x)$, $P_{ok}(x = 1)$, $G_{ok}(x = 0)$
2. $P_{req}(x = 1)$, $G_{req}(x)$, $P_{ok}(x = 1)$, $G_{ok}(x = 1)$
3. $P_{req}(x = 1)$, $P_{ok}(x = 1)$, $G_{req}(x)$, $G_{ok}(x = 0)$
4. $P_{req}(x = 1)$, $G_{req}(x)$, $G_{ok}(x = 1)$, $G'_{req}(x)$, $G'_{ok}(x = 0)$, $P_{ok}(x = 1)$
5. $P_{req}(x = 1)$, $P_{req}(y = 1)$, $P_{ok}(y = 1)$, $G_{req}(y)$, $G_{ok}(y = 1)$, $P_{ok}(x = 1)$

7. [10 points]: Which of the above histories are **not** linearizable? Please explain why.

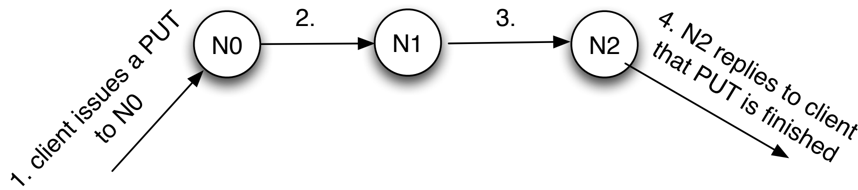

Figure 1: Alyssa P. Hacker's primary-backup replication scheme replicates data in a pipeline from the primary (N_0) to the two backups (N_1, N_2).

Alyssa P. Hacker comes up with a tweak on the traditional primary-backup system. In Alyssa's scheme, the primary still handles all PUT requests. However, instead of replicating PUT requests in parallel to N_1 and N_2, the replication is done in a pipeline from N_0 to N_1 and from N_1 to N_2. The steps of the protocol are shown in Figure 1.

- In step 1, a client issues a `put` request to the primary N_0. The primary processes the request locally.
- In step 2, the primary N_0 forwards the `put` request to backup node N_1 and N_1 processes the received request locally.
- In step 3, N_1 forwards the `put` request to backup node N_2 and N_2 processes the received request locally.
- Lastly, in step 4, N_2 replies to the client that its `put` request has finished.

If the primary N_0 fails, Alyssa plans to manually remove N_0 and resume the system with N_1 assuming the role of primary. We assume that N_1 and N_2 do not fail.

8. [5 points]:. In Alyssa's replication scheme, can the primary node N_0 handle `get` requests while guaranteeing linearizability? Explain. If not, which of the above histories are possible?

9. [5 points]:.In Alyssa's replication scheme, can clients randomly choose either N_1 or N_2 to issue each of its `get` request while guaranteeing linearizability? Explain. If not, which of the above histories are possible?

End of Quiz I