Computer Science Department

New York University

G22.3033-001 Distributed Systems: Fall 2010

Quiz I

All problems are open-ended questions. 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/20)	**II (xx/10)**	**III (xx/20)**	**IV (xx/15)**	**V (xx/10)**	**VI (xx/10)**	**Total (xx/85)**

Statistics:

Score range	number of students
>=60	2
50-59	7
40-49	1
30-39	8
<30	1

I Multiple choice questions:

Answer the following multiple-choice questions. Circle *all* answers that apply. Each problem is worth 4 points. Each missing or wrong answer costs -2 point.

A. Which of the following statements are true about the design of YFS and NFS?

1. A buggy NFS client program might corrupt the shared file system data structure.
2. A buggy YFS client program might corrupt the shared file system data structure.
3. There are many more different types of RPC messages between a NSF client and the NFS server than there are between a YFS client and the extent/lock server.
4. If only one YFS client is allowed to perform file system writes (e.g. create file, append data etc.) while all other YFS clients can only perform read-only file system operations, there is no need for using a lock service.

Answer: 2,3

B. Which of the following statements are true for sequential consistency (SC)?

1. SC is the strongest and yet practical consistency model.
2. SC allows for high availability during periods of network disconnectivity.
3. SC allows a client thread to read a stale value of some variable.
4. SC is only applicable to distributed shared memory systems and not applicable to distributed storage systems.
5. Suppose there are two concurrent write operations (executed by different threads): one changes variable A to 10, the other one changes variable B to 20. With SC, it's possible that one thread reads A=10, B=0 and another concurrent thread reads A=0,B=20 (assume both variables have zero as their initial value).

Answer: 1

C. Which of the following things are true for Bayou and Tra?

1. All storage replicas in Bayou and Tra eventually have identical state.
2. Bayou is able to detect update conflicts (i.e. when updates happened in a non-sequential manner.)
3. When using Tra, one must keep a replica for all data in the entire file system tree on every node.
4. One can and should build a banking application on top of Tra or Bayou.

Answer: 1

D. Which of the following statements are true?

1. Serializability requires the database manager to sequentially execute transactions, one after another.
2. Serializability demands that, if the database manager receives transaction A before transaction B, it will commit (or abort) transaction A before commiting (or aborting) transaction B.
3. Snapshot isolation is a multiversion concurrency control scheme that realizes serializability.
4. No multi-version concurrency control scheme can realize serializability; one must use 2-phase locking for serializability.

Answer: none

II Remote Procedure Call

After attending the lecture on failure recovery, Ben Bitdiddle becomes paranoid about failures. He is now concerned that the RPC library he completed in Lab 1 might not preserve at-most-once guarantee after the server reboots from a crash. After all, when the server recovers from a crash, it loses all its state about previous received RPCs and is at risk of executing a retransmitted RPC that it has already executed before the crash.

1. [10 points]: Does your completed RPC server in Lab 1 preserve at-most-once guarantee across reboots? If not, give a counterexample. If yes, please explain.

Hint: Recall that in our RPC library, each client creates a `rpcc` object and `bind()`s it to the corresponding server before using it. Further, the RPC packet header contains the following fields:

```
// add RPC fields before the RPC request data
req_header h(ca.xid, proc, clt_nonce_, srv_nonce_, xid_rep_window_.front());
req.pack_req_header(h);
```

Solution: The Lab1's template code preserves at-most-once guarantee across reboots. This is because each RPC server generates a random `srv_nonce_` upon initialization (refer to the constructor code for `rpcs`). Each client "binds" to the server by acquiring the server's `srv_nonce_` before sending the server any RPC requests. After the server fails and reboots, the newly generated `srv_nonce_` at the server would differ from that obtained by the client before the crash. When the server receives a RPC request whose `srv_nonce_` specified in the RPC header does not match its own, the server returns a failure (with failure code `rpc_const::oldsrv_failure`). Therefore, the rebooted server would never execute any duplicate RPC request that has already been executed by its previous incarnation.

III Threads and locks

Ben Bitdiddle wants to implement a reader-writer mutex (R/W lock) using ordinary pthread library. Ben has come up with the following simple implementation:

```
struct rwlock_t {
  int nreaders; //number of readers who have grabbed the R/W lock, initialized to 0
  int nwriters; //number of writers who have grabbed the R/W lock, initialized to 0
  pthread_mutex_t m;
  pthread_cond_t cv;
  rwlock_t() { //some initialization code here... }
};

void
read_lock(rwlock_t *rw) {
  pthread_mutex_lock(&rw->m);
  if (rw->nwriters > 0) {    <---- change "if" to "while"
    pthread_cond_wait(&rw->cv,&rw->m);
  }
  rw->nreaders++;
  pthread_mutex_unlock(&rw->m);
}

void
read_unlock(rwlock_t *rw) {
  pthread_mutex_lock(&rw->m);
  rw->nreaders--;
  if (rw->nreaders == 0) {
    pthread_cond_signal(&rw->cv);
  }
  pthread_mutex_unlock(&rw->m);
}

void
write_lock(rwlock_t *rw) {
  pthread_mutex_lock(&rw->m);
  if (rw->nreaders > 0 || rw->nwriters > 0) {  <---- change "if" to "while"
    pthread_cond_wait(&rw->cv, &rw->m);
  }
  rw->nwriters++;
  pthread_mutex_unlock(&rw->m);
}

void
write_unlock(rwlock_t *rw) {
  pthread_mutex_lock(&rw->m);
  rw->nwriters--;
  pthread_cond_signal(&rw->cv);<--change "pthread_cond_signal" to "pthread_cond_broadcast"
  pthread_mutex_unlock(&rw->m);
}
```

2. [10 points]: Ben has noticed multiple problems when testing the correctness of his R/W lock: 1) sometimes, there are multiple readers and writers simultaneously holding the lock, violating correctness. 2) sometimes, while multiple readers should be be able to simultaneously hold the lock in read mode, he found the code is causing them to complete the `read_lock` operation serially, one after another.

Please correct Ben's errors for him. (You may directly mark your corrections on the previous page.)

Solution: see previous page.

3. [10 points]: Alice P. Hacker is concerned that Ben's implementation is not fair to writers. In other words, readers might starve a waiting writer: there could be arbitarily many readers coming and holding the lock while a writer is waiting for the lock. Please give a R/W lock implementation that favors the writer. In other words, any waiting writer can grab the R/W lock after a bounded amount of time.

Solution: There are many possible ways to prefer writers. The simplest solution is to add an additional integer field, e.g. `num_waiting_writers`, to `rwlock_t`. In `read_lock`, the reader waits until both the `rw->nwriters` becomes zero and `rw->num_waiting_writers` becomes zero.

continued from previous page....

IV Crash Recovery

In the class, we have simply assumed that writing to a single disk sector is atomic with respect to failures. In other words, the write operation either succeeds in its entirety (leaving the new data) or not at all (leaving the old data intact). Ben Bitdiddle is using a disk for which this assumption does not hold. The disk may fail in the middle of an sector write, leaving partially written new data and corrupting existing data on that sector.

Ben would like to implement `atomic_write()` and `atomic_read()` on top of this disk. The correct behavior of these functions should be: if `atomic_write(addr, data)` succeeds, all subsequent `atomic_read(addr, buffer)` returns the new data written. If `atomic_write(addr, data)` fails because of a disk crash, all subsequent `atomic_read(addr, buffer)` returns the old data stored at $addr$ before the crashed `atomic_write` started.

4. [5 points]: Ben's first idea is to only write data in the first 500 byte of a 512-byte sector and reserve the last 12 byte as the checksum. Ben implements the `chsum_write(int addr, char[500] data)` function which writes both the 500-byte $data$ and its checksum, $chsum(data)$, into the underlying 512-byte sector with address $addr$. The `bool chsum_read(int addr, char *buffer)` function reads the underlying 512-byte sector and verifies that the last 12-byte checksum is correct for the preceding 500-byte data block. If so, the data is returned, otherwise a failure is returned. Can Ben directly use `chsum_write` and `chsum_read` for atomically reading and writing data blocks of 500-byte in size? Please explain.

Solution: `chsum_write` and `chsum_read` do not implement atomic write/read. This is because if disk fails in the middle of `chsum_write`, the sector contains partially written data, thus corrupting whatever data was on it before the write. Subsequently, `chsum_read` will always fail when reading that sector instead of returning either the valid old or new data.

5. [10 points]: Please help Ben design and implement the `atomic_write` and `atomic_read` functions. Hint: you might want to consider storing two copies of checksumed data at one address.

Solution: For every logical address `d`, store two copies of checksumed data at physical address `d0` and `d1`. Below is the pseudocode for writing and reading.

```
atomic_write(address_t d, char *data)
{
    if (chsum_read(d0, tmpbuffer) == valid) {
      chsum_write(d1,data)
      chsum_write(d0,data)
    } else {
      chsum_write(d0,data)
      chsum_write(d1,data)
    }
}

atomic_read(address_t d, char *buffer)
{
    if (chsum_read(d0, buffer) == valid) {
      return;
    } else {
      assert(chsum_read(d1,buffer)==valid);
    }
}
```

The `atomic_write` function keeps the invariant that *at least one of the two copies of data is valid under all circumstances*. You should be able to prove for youself that if the invariant holds true before `atomic_write`, then it stays true after either a successful execution of `atomic_write` or any arbitary crash in the middle. Since at least one copy of the data is valid, the `atomic_read` function simply reads that valid copy. If both copies are valid, `atomic_read` returns the first copy of the data.

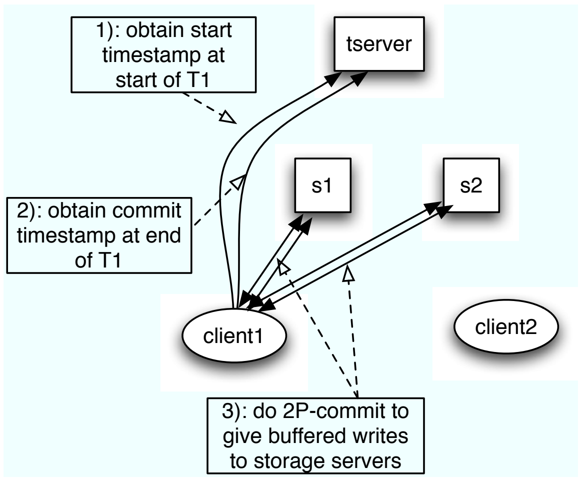
Only one of you got this question correct. A number of you write this `atomic_write` function instead:

```
atomic_write(address_t d, char *data)
{
    chsum_write(d0,data)
    chsum_write(d1,data)
}
```

Can you see what's wrong with this implementation(the `atomic_read` function is the same as before)?

V 2P commit and Snapshot isolation

Ben Bitdiddle wants to implement snapshot isolation in a distributed storage service. Ben's system consists of a single timestamp server ($tserver$) and a number of storage servers (e.g. $s1$ and $s2$) (see Figure below).


1): obtain start timestamp at start of T1
tserver
s1
s2
2): obtain commit timestamp at end of T1
client1
client2
3): do 2P-commit to give buffered writes to storage servers

Here's an example of how Ben's system works. Suppose $client1$ is executing a transaction T1 that writes two items X and Y. $client1$ first obtains a start timestamp for T1 from $tserver$, say, $T1.sts = 1$. During execution of the transaction, $client1$ buffers writes to X and Y locally. After T1 commits, $client1$ obtains a commit timestamp from $tserver$, say, $T1.cts = 10$. Since $s1$ is the server responsible for storing X and $s2$ is the server responsible for storing Y, $client1$ performs a 2P-commit (as the coordinator) with $s1$ and $s2$ to 1) check whether T1 suffers from write-write conflicts with other concurrent transactions and 2) write both items to $s1$ and $s2$ atomically if T1 can commit. In particular, $client1$ sends $T1.sts$ and $T1.cts$ as well as the buffered write of X to $s1$ in a 2P-prepare message. Likewise, it also sends $T1.sts$ and $T1.cts$ as well as the buffered write of Y to $s2$ as a 2P-prepare message. Both $s1$ and $s2$ check for write-write conflicts locally and vote accordingly.

Let's examine how an ongoing 2P-commit affects concurrent reads. For example, suppose $T1$ is in the middle of 2P-commit and $s1$ has just voted "yes" to commit $T1$'s write of X (but it has not known the final outcome of the 2P-commit yet). Suppose another concurrent transaction $T2$ wants to read X from $s1$ with start timestamp $T2 = 11$.

6. [10 points]: Which of the following action(s) is correct for $s1$ to serve T2's reads?

1. $s1$ returns the write of T1 (i.e. the version of X with timstamp 10) to T2 immediately.
2. $s1$ returns the latest version of X that's no bigger than T1's commit timestamp, e.g. suppose there's a version of X with timestamp 9, then $s1$ can return this version as the result of read to T2 immediately.
3. $s1$ blocks the read of T2 until it has received the outcome of the 2P commit.

Please explain your answer. Specifically, if you think a choice is incorrect, please explain why it is wrong.

Solution: Only 3 is correct.

1 is incorrect because the outcome of T1's commit is not known yet and the other participant $s2$ might have voted to abort T1. Therefore, it's wrong to return the write of T1 to T2 immediately.

2 is incorrect. There are two problems. First, snapshot isolation requires T2 (with start timestamp 11) to see the effect of all the committed transactions with commit timestamps earlier than 11. If T1 is to commit, T2 has to see the writes of T1. Therefore, before $s1$ knows the outcome of T1, it has to block the reads of T2. Second, if $s1$ were to return an older value of X (say, with version stamp 9), T2 might not even see a valid snapshot! In particular, T2 might have gotten an older version of X (with version stamp 9) and later, when T2 requests to read Y from $s2$ (and $s2$ has already received the 2P outcome from cooridnator), $s2$ will return the new value of Y (with timestamp 10) to T2. The older value of X and the new value of Y do not form a valid snapshot!

VI G22.3033-001

7. [5 points]: Describe the most memorable error you have made so far in one of the labs. (Provide enough detail so that we can understand your answer.)

We would like to hear your opinions about the class so far, so please answer the following two questions.

8. [3 points]: What is the best aspect of this class?

9. [2 points]: What is the worst aspect of this class?

End of Quiz I