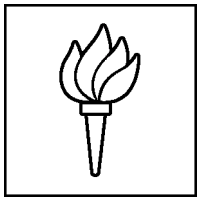


Computer Science Department

New York University

G22.3033-001 Distributed Systems: Fall 2009

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/25)	II (xx/10)	III (xx/25)	IV (xx/20)	V (xx/10)	Total (xx/90)

Statistics:

Score range	number of students
>=70	2
60-70	1
50-60	6
40-50	6
30-40	2
<30	2

I Remote Procedure Call

Ben Bitdiddle believes that at-most-once RPC is not necessary for implementing the extent service. *Ben uses the naive RPC library in Lab 1 with retransmission enabled, but without all the machinary that implements at-most-once execution.* He executes this following sequence of code to store and retrieve the extent with key `eid`: (You should assume there is only one yfs_client active in the system.)

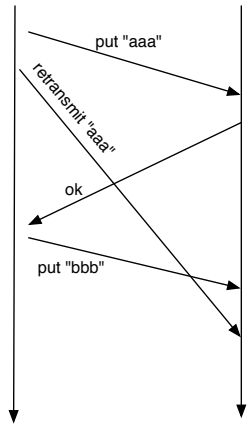
```
//cl is the rpcc object for communicating with the extent_server
ret = cl->call(extent_protocol::put,eid,"aaa",r);
assert(ret == extent_protocol::OK);
ret = cl->call(extent_protocol::put,eid,"bbb",r);
assert(ret == extent_protocol::OK);
ret = cl->call(extent_protocol::get,eid,buf);
assert(ret == extent_protocol::OK);
cout << buf << "\n";
```

1. [2 points]: What is the *expected* content of `buf` at the end of the above sequence of code? (You can assume the sequence of code is the only RPC client active in the system.)

Solution: The expected content of `buf` is ”bbb”.

2. [8 points]: What are the potential *unexpected* contents of `buf`? (Recall that the network may lose, duplicate or reorder packets. Furthermore, the naive RPC library simply retransmits any request for which it has not received any reply.) Draw a timing diagram to explain each unexpected content of `buf`. (Your timing diagram should contain a time line for both the client and the server as well as all the messages exchanged between them.) Can these unexpected outcomes occur with a RPC library that implements at-most-once delivery? Why?

Solution: `buf` may contain the unexpected contents ”aaa”. See timing diagram below.



With at-most-once delivery semantics, this unexpected result won’t happen. The client will not issue `put "bbb"` until `put "aaa"` has finished. Since `put "aaa"` is executed at most once, the later `put "bbb"` overwrites the value of eid with ”bbb”, ensuring the later `get` result to be ”bbb”.

Ben proposes to implement at-most-once RPC by having the RPC server keep an in-memory *replay* buffer (`replaybuf`). In particular, each RPC client generates a random 64-bit number as the RPC request identifier. Since the RPC identifier space is fairly large (64-bit), we can assume that the identifiers generated by all RPC clients are unique. The RPC server remembers each RPC request that it has seen in the `replaybuf`. If a received RPC request is already in the `replaybuf`, then the RPC server treats it as a duplicate request. In order to prevent the `replaybuf` from growing without bound, Ben also removes all entries that are present in the `replaybuf` for longer than t seconds based on the `first_seen` field. Ben's RPC client implementation is the same as his Lab1's naive implementation with retransmission enabled. The pseudocode for his rpc server implementation is as follows (locking is omitted, but you should assume Ben performs locking correctly):

```
struct rpc_entry {
  struct timeval first_seen;
  bool rep_present;
  marshall rep;
  rpc_entry() {
    gettimeofday(&first_seen, NULL);
    rep_present = false;
  }
};
void rpcs::dispatch(...)
{
  ...
  //replaybuf is a hash map of rpc_entry
  //reqid is the 64-bit identifier associated with the RPC request
  if (replaybuf.find(reqid) != replaybuf.end()) { //replaybuf contains reqid
    if (replaybuf[repid].rep_present) {
       return replaybuf[repid].rep to the RPC client
    }else{
       do nothing
    }
  }else {
     replaybuf[repid] = rpc_entry();
     execute the corresponding RPC handler
     add reply to replaybuf[repid].rep, set replaybuf[repid].rep_present to true
  }
}

//executed periodicaly in a separate thread
void rpcs::expiretimer(...)
{
    foreach repid in replaybuf {
         if ((now - replaybuf[repid].first_seen) > t)
        remove repid entry from replaybuf
   }
}
```

3. [5 points]: Does Ben's RPC implementation always guanrantee at-most-once execution? If your answer is no, please give concrete examples.

Solution: No. A previous executed RPC might be deleted from the replaybuf after t seconds. Thus, if a client retransmission occurs t seconds after the initial execution, the RPC server would execute the RPC more than once.

4. [10 points]: Suppose the amount of time it takes the network to deliver a packet in one way is bounded by δ seconds. Furthermore, assume the clock drift of between any pair of arbitary machines is at most ϵ seconds. In other words, if machine M1 observes that $x1$ seconds have passed based on its local clock, then any other machine (e.g. M2) will observe that $x2$ seconds have passed according to M2's local clock such that $x1 - \epsilon \leq x2 \leq x1 + \epsilon$. Under these two assumptions, describe how Ben's replaybuf-based RPC library can be made to guarantee at-most-once delivery in the face of lost, reordered, duplicate packets **and** client or server failures. (Your solution should still have retransmissions to deal with occasional losses and it should avoid writing to disk).

Solution: As we see from Problem 3, Ben's design lacks at-most-once semantics because the server might receive a duplicate RPC request for some RPC that the server has already deleted from `replaybuf`. Therefore, to guarantee at-most-once, we must ensure that the server will not see any duplicate for any deleted RPC request. Intuitively, such guarantee is achievable if the network does not delay packets indefinitely *and* if the client does not retransmit indefinitely. Let us demand that the client stop retransmitting a RPC request x seconds after it first sends the RPC request (the client RPC library returns an error for any RPC with no corresponding reply upon stopping retransmission.) The question is, how should we set x in order to guarantee that the server does not see duplicate RPCs?

When the server first receives a RPC request at time s (according to the server's local clock), what does it learn about when the client first sent the corresponding RPC request? Certainly, the client must have sent the RPC request no later than s, but could be as early as $s - \delta$ or even earlier if the first transmission was lost. Since the client will stop retransmission x seconds after the first transmission, the latest time the client can attempt its last retransmission will be $s + x$ according to the server's local clock (assuming no clock drift). With clock drift, the latest time the client attempts retransmission will be $s + x + \epsilon$. This last retransmission is guaranteed to arrive at the server at time no later than $s + x + \epsilon + \delta$. Therefore, if $s + x + \epsilon + \delta < s + t$, the server will be able to detect duplicate because it still holds the previously received RPC request. Therefore, we should set x to be no more than $t - \epsilon - \delta$.

We also need to make sure that when the server fails and recovers, it does not accidently execute duplicate RPC requests. This can be achieved if the rebooted server would return failures for all received RPCs for t seconds upon starting up.

The key insight is to realize that at-most-once execution is different from exactly-once-execution. With the former semantics, the RPC system has the freedom to simply return failure (e.g. see the at-most-once-failure return code in rpcc.cc in the labs). Thus, the client can give up an RPC whose reply it has never received in a bounded time and the server can return failure for an RPC whose execution status is unclear.

II Threads and mutexes

Ben Bitdiddle implements his Lab 1 lock server as follows:

```
1: int
2: lock_server::acquirereq(string lname, int &r) //RPC handler
3: {
4:    pthread_mutex_lock(&server_mutex);
5:    lock *l = locks[lname]; //assume lname already exists
6:    if (l->state != FREE)
7:     pthread_cond_wait(&server_cond, &server_mutex);
8:    l->state = LOCKED;
9:    pthread_mutex_unlock(&server_mutex);
10:   return lock_protocol::OK;
11:}
12: int
13: lock_server::releasereq(string lname, int &r)
14: {
15:   pthread_mutex_lock(&server_mutex);
16:   lock *l = locks[lname];
17:   l->state = FREE;
18:   pthread_mutex_unlock(&server_mutex);
19:   pthread_cond_broadcast(&server_cond);
20:   return lock_protocol::OK;
21: }
```

5. [5 points]: Ben finds the his lock_server can grant the same lock to multiple clients simultaneously. Identify and correct his mistakes. (You may directly modify Ben's code fragments. Please write correct code.)

Solution: Change the line 6 to `while (l->state != FREE)`.

Some students also propose changing line 19 to `pthread_cond_signal(&server_cond)`. This change is incorrect.

6. [5 points]: In `releasereq`, can Ben move line 19 to be in front of line 15? Explain.

Solution: No. Moving line 19 introduces the sleep-wakeup race. If the thread performing releasereq wakes up all other threads sleeping on `server_cond` before setting the lock state to be FREE, its broadcast signal is potentially "lost" because all the woken-up threads might see that the lock state LOCKED and go back to sleep.

III Crash Recovery

Ben Bitdiddle decides to store the content of his extent_server on the local ext3 file system of the extent_server. (Recall that the ext3 file system implements a redo logging scheme like that in Cedar, i.e. the ext3 file system logs the modified state for each file system meta-data operation and periodically flushes the log to disk.) Ben's RPC handlers for the extent_server are as follows:

```
int extent_server:::put(extent_protocol::extentid_t id, std::string buf, int &)
{
  //id2filename(id) converts the extent id into a unique filename deterministically.
  string f = id2filename(id);
  int fd = open(f.c_str(), O_CREAT|O_TRUNC|O_WRONLY);
  if (fd < 0) return extent_protocol:IOERR;
  int r = write(fd, buf.c_str(),buf.size());
  close(fd);
  if (r >= 0)
    return extent_protocol::OK;
  else
    return extent_protocol::IOERR;
}

int extent_server::get(extent_protocol::extentid_t id, std::string &buf)
{
  string f = id2filename(id);
  int fd = open(f.c_str(), O_RDONLY);
  if (fd < 0) return extent_protocol::NOENT;
  char *p = new char[MAX_EXTENT_SIZE];
  int n = read(fd, p, MAX_EXTENT_SIZE);
  if (n >= 0) {
    buf = string(p, n);
    return extent_protocol::OK;
  }else{
    return extent_protocol::IOERR;
  }
}
```

7. [10 points]: Ben uses his yfs_client to create two empty files named ”aaa” and ”bbb” one after another in the root directory. Suppose the extent_server process crashes in the middle of creating ”bbb” (and after the successful completion of creating ”aaa”). When Ben restarts his extent_server, what are the possible contents of his yfs root directory upon restarting the extent_server? Explain. (You should assume that there is only one yfs_client active in the system for all questions in this section.)

Solution: After the successful creation of ”aaa”, the file content corresponding to the root directory contains the entry for ”aaa”. When yfs crashes in the middle of creating ”bbb”, the extent_server could be in the midst of executing `put` where the contents of the `buf` argument contain dir entries for ”aaa” and ”bbb”. If the extent_server crashes before the `open` call, the recovered root directory would contain ”aaa” only. If the extent_server crashes after the `open` call, the receovered root directory would be empty (since the open call truncates the file content). If the extent_server crashes after the successful `write` call, the recovered directory would contain both ”aaa” and ”bbb”.

The above discussion assumes that it’s Ben’s extent_server process that has crashed and *not* the operating system (hence ext3 file system) that runs the extent_server process. If the OS that runs extent_server crashes in the middle of creating ”bbb”, the above three outcomes are still possible depending on when the ext3 flushes its log to disk.

8. [10 points]: Frustrated by the anomalies seen in the crash recovery of file creations, Ben asks Alyssa Hacker for help. Alyssa remembers that the local ext3 file system operation `rename(const char *oldpath, const char *newpath)` is an atomic operation, i.e. if failure occurs in the middle of this operation, the rename operation appears to either have happened or not at all upon recovery. Perform Alyssa's fix on Ben's code to ensure that Ben's yfs can recover correctly during the extent_server failure of creation operations. You can directly write on Ben code in the previous page.

Solution:

```
int extent_server:::put(extent_protocol::extentid_t id, std::string buf, int &)
{
  //id2filename(id) converts the extent id into a unique filename deterministically.
  string f = id2filename(id);
string tmpf = f + ".tmp";
  int fd = open(tmpf.c_str(), O_CREAT|O_TRUNC|O_WRONLY);
  if (fd < 0) return extent_protocol:IOERR;
  int r = write(fd, buf.c_str(),buf.size());
  close(fd);
rename(tmpf.c_str(), f.c_str());
  if (r >= 0)
    return extent_protocol::OK;
  else
    return extent_protocol::IOERR;
}
```

With this fix, if the extent_server crashes before the `rename` statement, the recovered root directory would contain "aaa" only. If the extent_server crashes during the `rename` statement, (being atomic, the rename either happens or not all) and hence the recovered root directory contains either "aaa", or "aaa" and "bbb".

9. [5 points]: Does Alyssa's fix ensure that Ben's yfs can always recover correctly during the failure of the extent_server process during arbitary yfs operations? Explain.

Solution: No. Alyssa's fix only ensures that the recovery is correct when crash happens in the middle of file creations (or unlinks) by making the `put` operation all-or-nothing atomic. For yfs operations that involve multiple `put` operations, e.g. renaming files across directories, there is no all-or-nothing atomicity guarantee across multiple `put` operations and hence crash in the middle of yfs rename operations might not be recovered correctly.

IV Consistency

Ben Bitdiddle plans to replicate his extent service in Lab 4 to include two servers: X and Y. His new extent_client implementation stores extents at both servers and reads extents from either X or Y at random. Other than extent_client, Ben keeps his Lab4 solution unchanged (Recall that in Lab4, Ben has enabled locking to cope with concurrent accesses to the file system state). The relevant changes in Ben's yfs_client to enable replication are as follows:

```
extent_protocol::status
extent_client::put(extent_protocol::extentid_t eid, std::string buf)
{
  extent_protocol::status ret = extent_protocol::OK;
  int r;
  //cl_X is the rpcc client bound to server X
  ret = cl_X->call(extent_protocol::put, eid, buf, r);
  //cl_Y is the rpcc client bound to server Y
  ret = cl_Y->call(extent_protocol::put, eid, buf, r);
  return ret;
}

extent_protocol::status
extent_client::get(extent_protocol::extentid_t eid, std::string &buf)
{
  extent_protocol::status ret = extent_protocol::OK;
  if (rand() % 2 == 0) { //randomly reads from X or Y
    ret = cl_X->call(extent_protocol::get, eid, buf);
  }else{
    ret = cl_Y->call(extent_protocol::get, eid, buf);
  }
  return ret;
}
```

> **10. [10 points]:** Does Ben's replicated extent service provide sequential consistency? For this and the rest of the questions, you should assume there is no failure. If your answer is no, give an example scenario involving yfs file system operations that cannot happen with a sequentially consistent extent service but could happen when using Ben's extent service and explain.

Solution: Yes. The new extent service guarantees sequential consistency. This is because the writes are executed by both extent_servers *one at a time* (and hence in the same order) because all writes are issued by yfs_clients while holding the corresponding lock from the external lock_server.

In general, when the writes are done without locking from the external lock_server (e.g. when one is using the extent service as a simple key-value store), this implementation does not guarantee sequential consistency because the different extent_servers might receive writes in different order and hence execute them in different order, violating sequential consistency.

I give both above answers full score as long as the explanation is clear.

Ben decides to tweak his implementation again so that all the modifications are sent to server X. Server X stores the extent and further forwards it to server Y. The extent_client still reads extent from either server X or Y at random. The new changes in Ben's code are as follows.

```
extent_protocol::status
extent_client::put(extent_protocol::extentid_t eid, std::string buf)
{
  extent_protocol::status ret = extent_protocol::OK;
  int r;
  ret = cl_X->call(extent_protocol::put, eid, buf, r);
  return ret;
}

extent_protocol::status
extent_client::get(extent_protocol::extentid_t eid, std::string &buf)
{
  extent_protocol::status ret = extent_protocol::OK;
  if (rand() % 2 == 0) { //randomly reads from X or Y
    ret = cl_X->call(extent_protocol::get, eid, buf);
  }else{
    ret = cl_Y->call(extent_protocol::get, eid, buf);
  }
  return ret;
}

//the RPC handler for extent_server
int
extent_server::put(extent_protocol::extentid_t eid, string buf, int &r)
{
  int ret = extent_protocol::OK;
pthread_mutex_lock(&extent_server_lock);
  in_memory_extents[id] = buf;
  if (me == "X") //if I am server X, forward the put request to server Y
   ret = cl_Y->call(extent_protocol::put, eid, buf, r);
pthread_mutex_unlock(&extent_server_lock);
  return ret;
}
```

11. [5 points]: Does Ben's new extent service achieve sequential consistency in the context of his yfs implementation (using a lock server)? If your answer is no, explain with an example.

Solution: Yes. When the external lock_server is used in combination with the extent service, all reads and writes to the extent service happen *one at a time* across all yfs_clients, ensuring sequential consistency.

12. [5 points]: If Ben intends to use his extent service as a generic key-value store (i.e. using it without an external lock server), does his key-value store implement sequential consistency? Explain. If your answer is no, explain with an example.

Solution: No. This implementation is better than that in Question 10 in the sense that all extent servers at least execute writes in the same order. However, reads might occur in the middle of some write w because of the lack of external lock service. Since reads do not go to the same extent_server, they might see inconsistent result depending on whether an ongoing write has finished at the extent_server the reads are performed or not. The slides of Lecture 8 give an example: a read might first go to an extent_server and see the effect of ongoing write w and a later read from the same client might go to a different extent_server and does not see the effect of ongoing write w, thus violating sequential consistency.

V G22.3033-001

13. [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.

14. [3 points]: What is the best aspect of this class?

15. [2 points]: What is the worst aspect of this class?

End of Quiz I