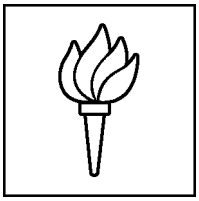

Computer Science Department

New York University

G22.3033-006 Distributed Systems: Fall 2008

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/15)	II (xx/10)	III (xx/30)	IV (xx/20)	V (xx/15)	VI (xx/10)	Total (xx/100)

I MapReduce

Ben Bitdiddle joins Google and gains access to a 1000-machine cluster. He is about to start his first MapReduce job of counting word frequencies among a collection of documents (128 TB in size). Ben recalls from distributed systems' lecture that the smaller map tasks are, the better the load balance is during Map phase. Therefore, Ben decides to configure his MapReduce job to divide the Map phase into pieces each with 64 KB. He also recalls from lecture that the final MapReduce output will be spread among R files, one for each reducer. Annoyed that the output is scattered among multiple files, he also decides to set $R = 1$ so all output resides in a single file.

1. [5 points]: Explain what is wrong with Ben's MapReduce configuration (It's preferrable to explain with concrete numbers). Give a reasonable MapReduce configuration. For each configuration parameter, explain briefly why yours is a reasonable value.

Ben is unhappy about the performance of his MapReduce job and decides to do some optimization. Since disks tend to be a bottlenecked resource in the shared cluster (due to extensive use by other jobs), he decides to eliminate unnecessary disk writes in the MapReduce library. He notices that there are two types of disk writes in MapReduce. First, each mapper writes intermediate output to its local disk. Second, each reducer transfers intermediate output from all mappers and writes them to its local disk before sorting them.

2. [5 points]: Ben proposes to eliminate both mapper-side and reducer-side disk writes. In Ben's proposal, each mapper immediately transfers intermediate results to all reducers who directly consume them without writing to disk at all. Assuming there are no worker failures, does Ben's proposal work? Why?

3. [5 points]: Ben makes another optimization proposal. This time, he decides to keep reducer-side disk writes, but proposes to eliminate mapper's disk writes by having each mapper directly transfer intermediate results to all reducers. Should the MapReduce team adopt Ben's proposal? Why?

II Threads and mutexes

Ben followed the instructions of the staff in lab 1, but is curious about the Lab 1's requirement to use a separate granter thread on the lock_server to grant locks to clients. He decides to do it differently without a separate granter thread. The sketch of Ben's new lock_client code is as follows:

```
//lock_client's lock is in one of 4 states: ABSENT, AQUIRING, FREE, LOCKED
lock_client::acquire(string lname)
{
  pthread_mutex_lock(&client_mutex);
  lock *l = locks[lname];
  while (l->state!=FREE) {
    if (l->state == ABSENT) {
      l->state = ACQUIRING;
      cl->call(lock_protocol::acquire, lname, id, granted);
      if (granted) break;
    }
    pthread_cond_wait(&l->cond, &client_mutex);
  }
  l->state = LOCKED;
  pthread_mutex_unlock(&client_mutex);
}
lock_client::release(string lname)
{
  pthread_mutex_lock(&client_mutex);
  lock *l = locks[lname];
  l->state = ABSENT;
  cl->call(lock_protocol::release, lname, id, dummy);
  pthread_mutex_unlock(&client_mutex);
}
lock_client::grantreq(string lname, int &r) //RPC handler
{
  pthread_mutex_lock(&client_mutex);
  lock *l = locks[lname];
  l->state = FREE;
  pthread_cond_signal(&l->cond);
  pthread_mutex_unlock(&client_mutex);
}
```

Ben's new server RPC handlers are as follows:

```
//lock_server's lock is in one of 2 states: FREE, LOCKED
lock_server::acquirereq(string client_id, string lname, int &r) //RPC handler
{
  pthread_mutex_lock(&server_mutex);
  lock *l = locks[lname];
  l->queue.push_back(client_id);
  if (l->state == FREE) {
    if (l->queue.front() == client_id) {
      r = true; //immediately grant client_id lock as acquire RPC returns
    } else {
      rpcc *cl = get_subscriber(l->queue.front());
      cl->call(lock_protocol::grant, lname , dummy);
```

```
    }
    l->state = LOCKED;
  }
  pthread_mutex_unlock(&server_mutex);
}

lock_server::releasereq(string client_id, string lname, int &r) //RPC handler
{
  pthread_mutex_lock(&server_mutex);
  lock *l = locks[lname];
  assert(l->state == LOCKED && l->queue.front() == client_id);
  l->queue.pop_front();
  if (l->queue.size() > 0) {
    rpcc *cl = get_subscriber(l->queue.front());
    cl->call(lock_protocol::grant, lname, dummy);
  }else {
    l->state = FREE;
  }
  pthread_mutex_unlock(&server_mutex);
}
```

4. [10 points]: Now the system deadlocks once in a while. Show a scenario that results in a deadlock. (Draw a message time diagram and explain. Use a time line for each client and server involved. Show labeled arrows between the lines for each message.)

III Logging

After attending the crash recovery lecture, Ben Bitdiddle decides to add redo-logging to his lock server in Lab 1 so that it can recover from crashes. Ben has coded his lock server according to the Lab 1's instructions. Below is the relevant code segment of his new modified lock_server. The function `log_operation` appends the triple (op_name, client_id, lname) to a log file stored on the server's local disk. It has its own internal lock structure to be thread safe. Other than the log file, Ben's lock_server keeps no persistent state on disk.

```
//lock_server's RPC handler for acquire RPC
lock_server::acquirereq(string client_id, string lname, int &r)
{
  pthread_mutex_lock(&server_mutex);
  lock *l = locks[lname];
  l->queue.push_back(client_id);
  if (l->state == FREE)
     pthread_cond_signal(&granter_cond);
  pthread_mutex_unlock(&server_mutex);
  log_operation(LOG_ACQUIRE, client_id, lname); //Ben's addition
}

//lock_server's RPC handler for release RPC
lock_server::releasereq(string client_id, string lname, int &r)
{
  pthread_mutex_lock(&server_mutex);
  lock *l = locks[lname];
  assert(l->state == LOCKED && l->queue.front() == client_id);
  l->queue.pop_front(client_id);
  l->state = FREE;
  if (l->queue.size() > 0)
     pthread_cond_signal(&granter_cond);
  pthread_mutex_unlock(&server_mutex);
  log_operation(LOG_RELEASE, client_id, lname); //Ben's addition
}

//lock_server's granter thread
lock_server::granter()
{
  pthread_mutex_lock(&server_mutex);
  while (1) {
    pthread_cond_wait(&granter_cond, &server_mutex);
    for (list::iterator iter = locks.begin(); iter!=locks.end();
        iter++) {
      lock *l = (*iter);
      if (l->state == FREE && l->queue.size()>0) {
        string client_id = l->queue.front();
        rpcc *cl = get_subscriber(client_id);
        l->state = LOCKED;
        cl->call(lock_protocol::grant, l->name, dummy);
        log_operation(LOG_GRANT, client_id, l->name); //Ben's addition
      }
    }
  }
  pthread_mutex_unlock(&server_mutex);
```

```
}
```

5. [10 points]: Write the corresponding recovery procedure for Ben. (You do not have to write syntactically correct C++ code. Pseducode is fine.) Write down the necessary assertions about the state of the lock server during recovery if you can.

During the testing of the new lock server (with failures and recovery), Ben has noticed many anomalies. First, he noticed that sometimes the LOG_GRANT record in the log file *precedes* its corresponding LOG_ACQUIRE record, which causes incorrect recovery. Second, after recovery, sometimes the lock_server would receive a release RPC for a lock whose the corresponding lock state is FREE, causing the assertion in release RPC handler to fail. Ben turns to Alice for help. Not only Alice fixes Ben's code, she also explains to Ben exactly why these anomalies happen.

6. **[10 points]:** Perform Alice's code fix. (You can directly mark on the psuedo-code on Page 6.)

7. **[10 points]:** Describe two scenarios that cause the two anomalies to happen.

IV Remote procedure call

After correcting his logging lock_server, Ben Bitdiddle notices that the RPC call at lock_client fails when the recovered lock_server comes back online. Ben quickly figures out that this is because the client's `rpcc` object is bound to the crashed lock_server instead of the newly recovered lock_server. Ben's fix is to delete the existing `rpcc` object, create and bind a new `rpcc` object, and re-try the RPC call. For example, the code below show the old and new acquire() function after Ben's fix. Similar fix is applied to the release() function.

```
//The old acquire() function
lock_client::acquire(string lname)
{
   ...
   ret = cl->call(lock_protocol::acquire, id, lname, dummy);
   assert(ret >= 0);
   ...
}

//The new acquire() function with Ben's fix
lock_client::acquire(string lname)
{
   ...
   while (1) {
      ret = cl->call(lock_protocol::acquire, id, lname, dummy);
      if (ret >= 0) break; //RPC call is successful
      delete cl;
      cl = new rpcc(dstsock); //create a new rpcc object
      cl->bind(); //bind to the recovered lock_server
   }
   ...
}
```

8. [5 points]: Using Ben's new lock_client, show a scenario in which the server sees a duplicate request for lock "a" from the same client. (Draw a message time diagram. Use a time line for each client and server involved. Show labeled arrows between the lines for each message and indicate server failures if there is one.)

9. [10 points]: After realizing that he will have to deal with duplicate requests anyway, Ben conjectures that the *at-most-once* delivery feature in the current RPC library is useless and removes all code implementing this feature from the RPC library. Without *at-most-once* delivery, show a scenario in which the server gives lock "a" to two different clients, violating the goal that only one client should have the lock at any time. (Draw a message time diagram.)

10. [5 points]: Using *at-most-once* RPC and Ben's fix (which causes duplicate requests), can you come up with a scenario in which the server gives lock "a" to two different clients? Briefly explain.

V Consistency: Using DSM for extent service

Inspired by the distributed shared memory (DSM) system described by Li and Hudak in ”Memory Conherence in Shared Memory Systems”, Ben decides to take a drastically different approach to implement the extent service. In particular, he decides to directly use DSM as an extent service where each extent is treated like a DSM page. For his DSM-based extent service, Ben uses the centralized manager algorithm described in Section 3.2. In his implementation, each yfs_client acts as a DSM node and there is a separate manager node that keeps track of the ownership of pages (i.e. extents) among different yfs_clients.

11. [5 points]: Ben is happy that his new implementation passes the Lab 3 test which involves reading, writing and sharing files among multiple yfs_clients sequentially. Since DSM implements *sequential consistency*, where all concurrent read and write requests are executed according to some total order, there is only one node allowed to modify a particular extent at any given time. Therefore, Ben argues that there is no need to use the lock service at all for his lab 4 implementation. Will he be able to pass Lab 4 tests? Why?

Ben decides to modify the DSM-based extent service to get better performance. The relevant portion of the pseudocode is in the following fragment from Section 3.2:

```
Write fault handler:
  lock(ptable[p].lock);
  IF I am manager THEN BEGIN
    receive page p from owner[p];
  ELSE
    ask manager for write access to p;
  invalidate(p, ptable[p].copyset);
  ptable[p].access = write;
  ptable[p].copy_set = {};
  unlock(ptable[p].lock)
```

Ben changes the implementation of `invalidate`. The implementation in the paper returns from the call to `invalidate` after the node has received acknowledgements from all of the computers listed in `copyset`. Ben modifies `invalidate` to return as soon it has sent off the invalidation messages to all yfs_clients in `copyset`. His hope is to get better performance because the write fault handler doesn't have to wait until the acknowledgements on the invalidate messages have been received.

12. [10 points]: Does Ben's change preserve the *sequential consistency* property of his DSM? Suppose files F1 and F2 both contain "xxx" initially. Client 1 writes "aaa" to file F1 while client 2 *simultaneously* writes "bbb" to file F2. Under the original DSM design, is it possible that client 1 reads "xxx" from F2 after writing "aaa" to F1 *and* client 2 reads "xxx" from F1 after writing "bbb" to F2? Could it happen with Ben's modification? If so, give a scenario when this happens.

VI G22.3033-006

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