Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.824 Distributed System Engineering: Fall 2007

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.

Write your name on this cover sheet AND at the bottom of each page of this booklet.

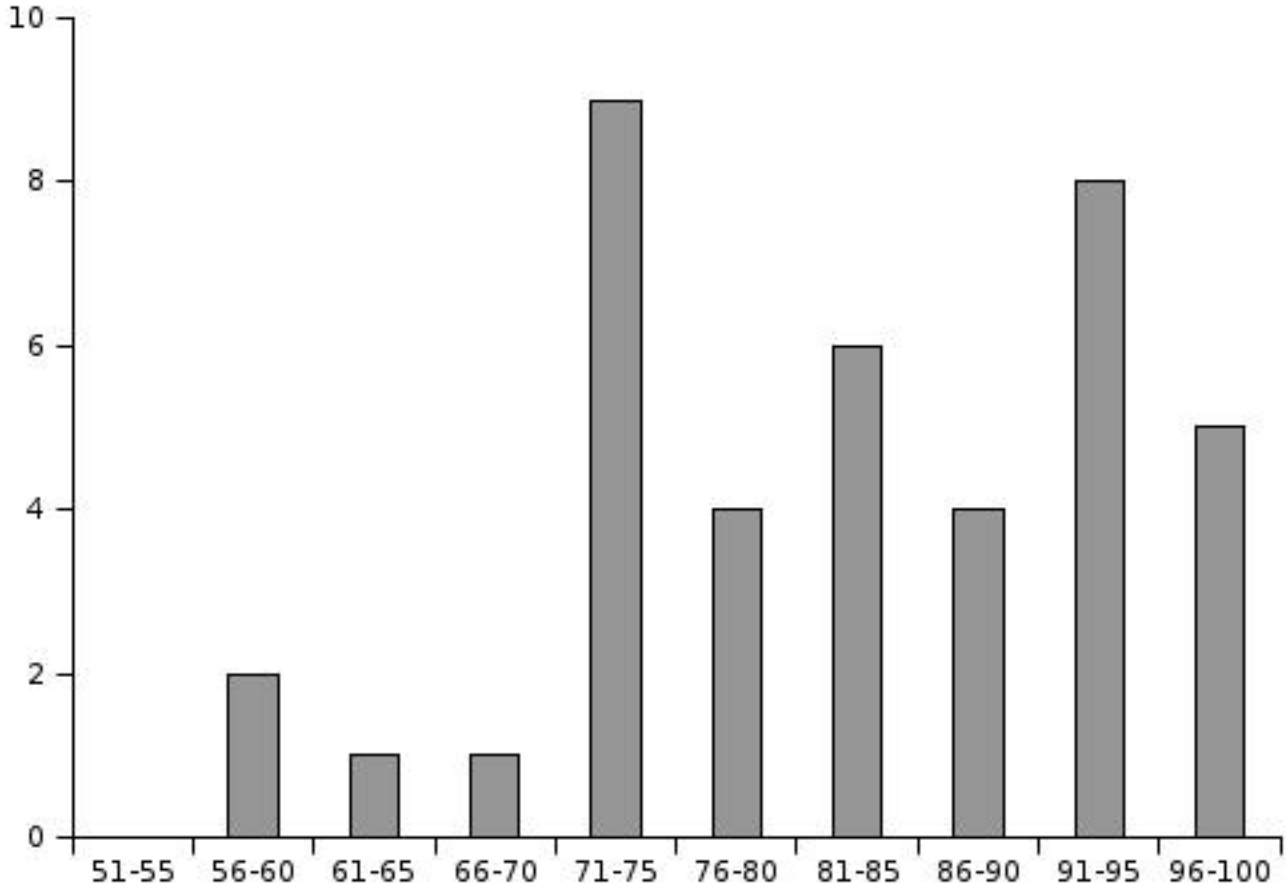
Some questions may be much harder than others. Read them all through first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to write down any assumptions you make. Be neat. If we can't understand your answer, we can't give you credit!

THIS IS AN OPEN BOOK, OPEN NOTES QUIZ.

I (xx/25)	II (xx/10)	III (xx/10)	IV (xx/15)	V (xx/15)	VI (xx/15)	VII (xx/10)	Total (xx/100)

Name:

Grade histogram for Quiz 1



I Remote procedure call

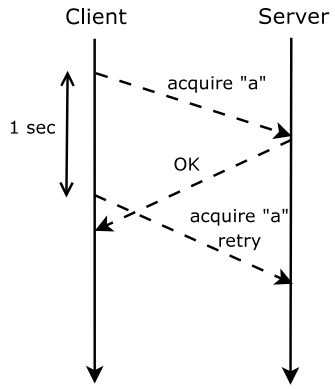
Ben finds writing programs with the YFS RPC library painful. After struggling with sequence numbers in lab 1 (lock server) and lab 5 (caching lock server), Ben wonders if he could get rid of sequence numbers (and the corresponding code) if the RPC library sends all RPCs over TCP instead of UDP. TCP is a reliable transport protocol that delivers packets to their destination even when the network may lose packets, and delivers all packets of a connection in order.

Ben modifies lock client, server, and YFS RPC library as follows:

- He removes the code that handles sequence numbers from client and server (i.e., the code to detect out of order, duplicate, etc. messages);
- He removes his sequence numbers as arguments from `cl.call` calls (where `cl` is an `rpcc` object);
- He changes the implementation of `cl.call` to start a TCP connection to the destination, if no TCP connection to the destination exists;
- `cl.call` sends the request over the TCP connection;
- He keeps the retransmission code so that the library retransmits after a second or so to recover from potentially failed TCP connections (e.g., if the destination fails or there is long enough network partition). The retry code will keep attempting to connect and set up a new TCP connection to the destination until the source has received the RPC response from the server.

1. [10 points]: 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 involves, and show labeled arrows between the lines for each message.)

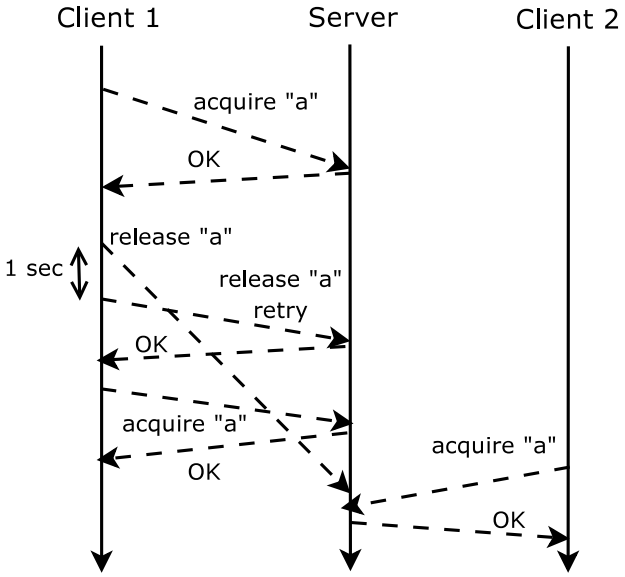
Any situation is fine that involved a packet being delayed long enough in the network to cause YFS RPC to start a new TCP connection and retransmit. The simplest, and most common, solution:



Name:

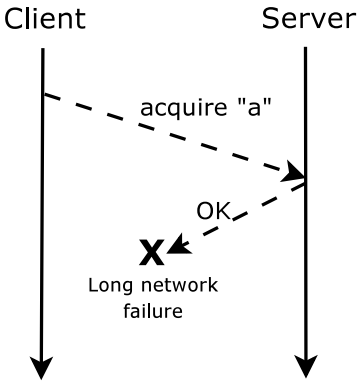
2. [10 points]: Show a scenario in which the server gives lock "a" to two different clients, violating the goal that only one client should have the lock at any time. (Draw a message time diagram.)

The most straightforward solution is to have a delayed release reach the lock server after a second acquire, falsely making the lock server think the client had released its lock, freeing it up for another client. The network or the operating system must delay the original release long enough for YFS to create a new TCP connection and resend the release. Note that the diagram below assumes a non-caching lock server, though you can make it work with the caching lock server as well.



3. [5 points]: Frustrated, Ben removes the retransmission code from the RPC library too, completely relying on TCP for retransmission. But, now he observes that the client can get into a state where it cannot successfully complete its RPC (i.e., it receives no response from the lock server). Show a scenario that results in this behavior. (Draw a message time diagram.)

Without RPC retries, any long network failure will cause TCP to give up and fail the connection, returning -1 to the lock client's call. For example:



Name:

II Threads and mutexes

Ben used fine-grain pthread locking in lab 5, but otherwise followed the instructions of the staff. After 60 hours of debugging a concurrency hell, he throws away his fine-grained locking implementation and switches to very coarse-grain locking. The sketch of his new client code is as follows:

```
// Release l, and send the lock to the server if the lock has been revoked
release(lock l)
{
  pthread_mutex_lock(&client_lock);
  ...
  if (revoked[l])
     cl.call(server, RELEASE, ...)
  ...
  pthread_mutex_unlock(&client_lock);
}

// A revoke request from the lock server, which sets revoked for lock l
revokereq(lock l)
{
  pthread_mutex_lock(&client_lock);
  ...
  revoked[l] = true;    // tell client that it should release lock l
  ...
  pthread_mutex_unlock(&client_lock);
}
```

The relevant server code is as follows:

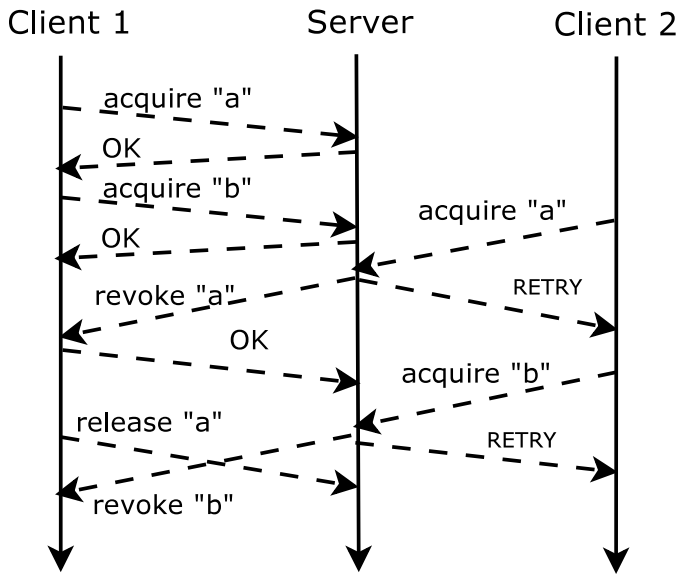
```
// A client asks for lock l
acquirereq(lock l)
{
  pthread_mutex_lock(&server_lock);
  ...
  if (taken[l])
    cl.call(holder_of_lock, REVOKE, ...);  // tell holder to release lock l
  ...
  taken[l] = true;    // a client has taken lock l
  ...
  pthread_mutex_unlock(&server_lock);
}

// A client asks to release l
releasereq(lock l)
{
  pthread_mutex_lock(&server_lock);
  ...
  taken[l] = false;    // client releases lock l
  ...
  pthread_mutex_unlock(&server_lock);
}
```

Name:

4. [10 points]: Now the system deadlocks once in a while. Show a scenario that results in a deadlock. (Draw a message time diagram.)

There are a few ways by which the system can deadlock. One good way is for a release RPC and a revoke RPC to cross paths. The server will be holding its lock waiting for the response for the revoke, while the client will be holding its lock waiting for the response to the release. Neither will be able to process the incoming RPC until it gets the response for its outstanding RPC (which will of course never be processed).



Name:

III Peer-to-peer: lookup

The pseudocode for Chord's join implementation, from figure 7 of the paper "Chord: a scalable peer-to-peer lookup service for Internet applications", is as follows:

```
n.join(n1)
  predecessor = nil;
  successor = n1.find_successor(n);

n.stabilize()
  x = successor.predecessor;
  if (x in (n, successor))
    successor = x;
  successor.notify(n);

n.notify(n1)
  if (predecessor is nil or n1 in (predecessor, n))
    predecessor = n1;
```

Ben observes that the Chord code for stabilization uses an indirect method to set the predecessor of a node, and proposes the following modifications to this pseudocode:

```
n.joinnew(n1)
  successor = n1.find_successor(n)
  if (successor is not nil)
    predecessor = successor.predecessor;
    if (predecessor is not nil)
      predecessor.set_successor(n);
    successor.set_predecessor(n);
```

`joinnew` replaces the existing join implementation from the paper, and Ben removes `stabilize` and `notify`.

5. [10 points]: Why is this change a bad one? (Give a brief explanation and a scenario that illustrates your explanation.)

If nodes with close node IDs join at the same time, only one might appear in the succ ring. For example, if a Chord ring consists of nodes 1 and 4, and nodes 2 and 3 join the ring concurrently, it's possible for both new nodes to have node 4 for a successor, and node 1 for a predecessor, and never find out about each other. In addition, if node leave the system, the ring doesn't repair because Ben deleted stabilize.

Name:

IV Peer-to-peer: data distribution

6. [5 points]: Explain briefly the purpose of Bittorrent's choke algorithm.

To implement a tit-for-tat scheme so that nodes download at a rate proportional to their upload rate.

7. [10 points]: If there are a few high-upload peers and many low-upload peers, does Bittorrent achieve the lowest total download time (across all peers)? If yes, give a brief argument why. If no, give a modification to the Bittorrent protocol that is likely to reduce the total download time.

No. The choke algorithm will make it likely that high upload peers match up with other high upload peers, and low upload peers with low upload peers. The high upload peers will finish quickly and then are likely to leave the system. Several fixes are possible: incentive high upload peers to stay online, remove penalty for low-upload nodes, etc.

V Consistency

Ben runs the following program on a computer with two processors and a single shared memory (with in-order execution of memory operations and no caches):

```
processor 0:
   x = 1;
   if(y == 0)
     critical section;

 processor 1:
   y = 1;
   if(x == 0)
     critical section;
```

On his computer at most one processor enters the critical section, as it should.

8. [5 points]: Ben ports the program to the distributed shared memory (DSM) system described by Li and Hudak in "Memory Coherence in Shared Virtual Memory Systems". What consistency model is implemented by DSM that makes this program work correctly? "Correctly" in this question means if two computers share `x` and `y` through DSM only one computer enters `critical section`. (Explain your answer briefly.)

DSM provides sequentially consistency. An implementation of a sequentially-consistent memory must ensure that there is some total order for all concurrent read and write operations that is also consistent with the order of the operations on a given processor. There is no total order that can result in computer 0 and 1 both entering the critical section.

Ben modifies DSM slightly to get a bit more parallelism. The relevant portion of the pseudocode is 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 it 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 computers 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.

9. [10 points]: Does Ben's program still run correctly? (Briefly explain.)

No. For example, consider the case that processor 0 and 1 have both a read-only copy of the pages that contain x and y, and that x and y are on different pages. After processor 0 updates x, it may take a long time before processor 1 finds out about the update and processor 0 doesn't wait until processor 1 acknowledges that it has received the update. In the mean time processor 1 can get ownership of the page that contains y, update y, and then read its out-of-date copy of the page that contains x. The update of y might not arrive at processor 0 before it has read an out-of-date copy of the page that contains y. Processor 0 will read 0 for x and processor 1 will 0 for y, and both will enter the critical section.

Name:

VI Caching in YFS

When running YFS with the caching lock server and extent server (lab 6), an alert 6.824 student observes the following communication pattern often. When two clients are sharing the same file "f", YFS transfers the extent for file "f" between the two clients through the extent server. That is, when client 1 has modified "f" and client 2 wants the file, client 1 writes the extent on the release of the lock for file "f" to the extent server. When client 2 receives the lock for file "f" from the lock server, it reads the extent for file "f" from the extent server. The extent for file f is transferred twice over the network, and when "f" is large, the student observes long delays before client 2 receives the extent for "f".

10. [15 points]: Describe a small, simple set of changes to the YFS protocols used in lab 6 so that YFS transfers the extent for "f" only once over the network (i.e., directly from client 1 to 2). Your modified protocol should pass all the tests for lab 6 and work under lossy conditions (i.e., the network may lose packets). Make sure to specify the RPCs in your modified protocol, their arguments, and their results.

There were several possible correct solutions to this problem. One very simple one is to store ***pointers*** *to extents at the extent server, rather than the extents themselves. Thus, on a* `dorelease()/flush()`*, the YFS server will* `put()` *its own hostname:port on the extent server, instead of the extent itself, and keep the extent locally cached. Then a* `get()` *call from another client fetches the hostname:port, and then uses a new RPC (say* `get_cached_extent`*) to retrieve the extent directly from the client. Though this adds an extra round trip time, it doesn't involve modifying the lock protocol. This assumes clients never fail.*

- `put(extentid_t id, int seqno, std::string &client)`
- `get_cached_extent(extentid_t id, int &dummy)`

Note that the sequence number is necessary in the put to avoid a duplicated delayed packet in a lossy network overwriting the put from a newer extent owner. The usual extent-locking convention ensures that a `get()` *always see the most recent writer's hostname:port.*

Another workable solution involves modifying the lock protocol to send extents directly to the new lock owner as part of the `dorelease()/flush()` *method. In this solution, the lock server adds the hostname:port of the new lock holder in the* `revoke()` *RPC to the current lock holder. The current lock holder must then remember this address, and pass it to* `flush()` *when the lock is released. Then it sends a new RPC (say* `put_cached_extent()`*) to the new lock holder's YFS server, which must add the extent to its cache. After that RPC is complete, the current lock holder can release the lock back to the lock server, which can grant it to the new lock holder. Once the new owner receives the lock, it is guaranteed to have the current version of the extent in its cache. Note that there should be a sequence number attached to the new RPC to avoid problems with delayed puts.*

Name:

VII 6.824

11. [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 6.824, so please answer the following two questions. (Any relevant answer will receive full credit!)

12. [3 points]: What is the best aspect of 6.824?

13. [2 points]: What is the worst aspect of 6.824?

End of Quiz I

Name: