Stanford University
Computer Science Department
CS 240 Quiz 2
Spring 2003

May 15, 2003

This is an open-book exam. You have 50 minutes to answer seven questions. Write all of your answers directly on the paper. Make your answers as concise as possible. Sentence fragments ok.

NOTE: We will take off points if a correct answer also includes incorrect or irrelevant information. (I.e., don't put in everything you know in hopes of saying the correct buzzword.)

Question	Score
1 - 2	
3 - 4	
5 - 6	
7	
total	

Stanford University Honor Code

In accordance with both the letter and the spirit of the Honor Code, I did not cheat on this exam nor will I assist someone else cheating.

Name and Stanford ID:

Signature:

Answer the following seven questions and, in a sentence or two, say *why* your answer holds. The first six questions are worth 5 points each, the last 10 points.

1. Assume you can buy one of (1) infinitely fast disk, (2) infinitely fast network, or (3) infinitely fast CPU. Give an ordering of most to least important for the webservers and workloads described in the FLASH paper Figures 9 and 10.
   This will depend on their and assumptions. Different answers are fine as long as their assumptions are reasonable. They only need to give general trands rather than specific comparisions.

   The FLASH paper is about throughput rather than latency. So the most important thing for "all cached" is whether the CPU or network is the bottleneck. It's hard to tell exactly, but as long as the average file size is more than 50KB, then the graphs for FreeBSD look CPU bound (since they are lower numbers). Similarly for Solaris.

   For all uncached: the main bottleneck is disk (3x), then perhaps CPU, network should be last.

2. Will a guest OS be more or less susceptible to livelock when running on VMware? Give two examples where the VMM can or cannot detect when a guest OS is livelocked or overloaded.
   The guest will effectively runs slower, so will be more suceptible to livelock.

   Some ways the VMM can tell when the guest is overloaded:

   (a) If the virtual network receive interface fills up.

   (b) If it can tell that the application starves.

   (c) If it knows what the guest system is supposed to do and it isn't doing it. E.g., if the guest is a software router, about as many packets as go in should come out.

   Without guest-specific knowlege, the VMM will not have much visibility into what is happening – e.g., it will be hard for it to tell if the packets are being dropped inside the guest on various queues.

3. Suppose Jimbo wants to test an NFS server for livelock and runs a client with this loop:

   ```
   while(1){
     send NFS READ RPC;
     wait for response;
   }
   ```

   Is the NFS server probably subject to livelock? What happens as Jimbo increases the number of client machines running this loop?
   This loop cannot livelock NFS, since it is subject to feedback (flow-control): does not do another request until it gets an ack. In fact, if we take the loop literaly, it does not retransmit when a packet is lost. Thus it is impossible for it to livelock the machine, since the first lost packet fill shut it up.

If the loop did retransmission, then as the number of clients goes to infinitity the probability of livelock goes to 1. Without retrans it maybe a able to instananously cause packet lots but the load would immediately go low.

4. Does the end-to-end argument imply that lower levels can do a bad job?
No. They must be at least as reliable as the upper layer. And, in reality, they should probably be a bit more reliable if they are built by different groups since otherwise the system may be quite difficult to reason about.

5. You start sending NFS operations over a reliable, in-order, duplicate-suppressing network stream (e.g., TCP). Jimbo claims that this will fix all the non-idempotent problems of using NFS over UDP. Is this true? Give your intuition or a concrete counter example.
*It will ensure that each request is done at most once, which fixes the problems with non-idempotent operations *until* the server crashes. However, if the client sends a request X and the server crashes before sending an ack, then the client has no way of knowing if X was performed or not.*

6. In LBFS, is the "size" returned by `GETHASH` redundant? What about the "count" taken by `CONDWRITE`? Give your reasoning why or why not.
GETHASH: if the returned SHA hash is in the database then the size is redundant — we assume the SHA cannot practically collide, thus the entry in the database can be trusted, including the size. However, if it is not in the database then we need the size so we can compute offsets for the subsequent READ requests. For CONDWRITE, the count is always redundant for similar reasons: since if the chunk is in the database with the given hash, then we know and can trust the size.

7. (10 points) You notice that leases inherently involve request-response messages. You decide to try to save network traffic by building leases on top of UDP and doing your own retransmission by setting a timer and retransmitting if you didn't get the appropriate reply. When the server grants a read lease, what should it do on retranmission? If a client gets a write revocation for a lease it does not hold, what should it do? Are there other interesting cases?
The big problem here is what happens if messages get reordered and retransmitted.

   (a) *In the first case: when you retransmit, you may want to push the lease expiration term out to give the same absolute amount of time. However, this opens the possibility that the client will receive multiple conflicting read leases: e.g., if its acks are all lost, it will keep receiving leases with expirations further and further into the future.*

   If you do not implicitly grant extensions on retransmission then as the term end gets closer and closer, you will eventually have to stop sending (or revert to 0 term lease). Additionally, you'll have to figure out what to do when issuing a revocation on a read lease that has not been acknowleged.

   (b) *Messages can get lost or reordered with retransmissions. For example, the server could send a message granting a read lease and then immediately send a write* *revocation. The revocation could arrive before the read lease, or the read lease message could be lost. The simplest thing: (1) attach a sequence number to each message (2) ack the write revocation and (3) make a note to discard any subsequent read lease with a lower sequence number.*

(c) *You'll have to see what they come up with for more subtle issues. There probably are some.*