Stanford University
Computer Science Department
CS 240 Sample Quiz 2 Answers
Spring 2004

May 14, 2004

These were from open-book exams. In general you had 50 minutes to answer 8-10 out of 10-12 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.)

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 **10 of the following 12 questions** (i.e., skip two) and, in a sentence or two, say *why* your answer holds. (5 points each).

1. *Three possible reasons: (1) not all clients need this functionality, so should be able to turn it off; (2) on a local system could still crash and lose data after file system operation returns to application, so having this extra guarentee is pointless in some situations. (3) when the NFS server replies to the client it does so without ensuring that the data was written to disk successfully.*

   Five points where given if you mentioned one of the three reasons. Three points where given if you had an answer that resembled one of the solutions but weren't specific enough.

2. *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.*

3. *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. *Problem is multiple sources of starvation:*

   (a) *If eth1 is busy, eth2 will starve.*

   (b) *If receive busy, send will starve*

   (c) *If net busy, app will starve.*

   Starvation = a scheduling problem where (1) a job can run too long or (2) some set of jobs are consistently selected over others. To solve the *first problem we give each job a time-slice (in this case a quota of the number of packets we're willing to let them process). To solve the first problem we round robin (fair) schedule between all different jobs.*

5. *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. *New connection requests are not flow-controlled. Additionally, poor video performance will causes people to push "reload," which causes more connection requests, which are not flow-controlled. And people could always launch denial-of-service attacks. We also accepted the answer that video streams are almost always UDP, so even though the initiation of a stream might be through TCP, the stream could be use UDP, and thus no (or poor) flow control can lead to livelock.*

7. *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.*

8. *If the server does not write its state to disk, then if the server crashes while a client program is doing a sequence of operations the file system state will (for example) go back in time by losing some updates. They mention that the client would otherwise have to detect server crashes and reply its modifications.*

9. *"Reasonable" is open to interpretation. Several ways:*

   (a) *Flush writes not on close but based on time (e.g., every 30 seconds).*

   (b) *Migrate ownership of the file on close so that the client owns it. (Sending ownership is cheaper than sending file data). This makes you vulnerable to clients going down and you must trust them.*

   (c) *Coda semantics: update file some point later in time (when your laptop reconnects) and do conflict resolution as necessary. Reeally cheap, but may require you manually fix your files.*

10. *No. For example, if the server crashes in the middle of a non-idempotent operation, it still must be able to recover from the crash to retain a consistent state.*

11. *No livelock; -1 point if you didn't give a reason why. Underutilization is a problem.*

12. *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.

13. *A defined state: all clients see the same valid record. The key feature is that all replicas must have the same state so that it does not matter where you do a read from. If all replicates play mutations in the same order and all the replicas you can read from are not stale, then you (and your friends) will always see the same value. It's not really correct, since clients cache replicas and so can read from a stale one before it gets shutdown.*

14. *NFS operations:*

    - *Idempotent: null, lookup, getattr, setattr, read, write, reader, statfs*
    - *not: create, remove, renam, link, symlink, mkdir, rmdir,*

    So less than half. Different semantics: you do a mkdir, acknolegement gets lost, you redo mkdir and get an error code.

15. *Unfortunately, since this question is so fuzzy it has more than one plausible answer. The most straightfoward answer is that he's half right. It is reasonable to push TCP out of the kernel into application*

space. However, doing so doesn't mean the app has to implement it: the reasonable way to do it is to put it in a library that the application can link into. Pushing functionality to the ends doens't mean they have to build it, just that they can (and can most likely select from an implementation.)

16. *Inode: live if the inode map points to it. Data block: live if it is contained in a file.*

17. *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.*

18. *For read: before sending the hash values back. Otherwise the client would use bogus values. The client similarly checks after it gets the gethash. For writes: the client checks before sending hashes, and the server rechecks before telling the client which ones it has. In both cases if they did not check, could get bad values.*

19. *Suspends receive processing; if the packet was not for screend you are not happy.*

    Two points were given for correctly describing the hack. The rest of the points were dependent on the description of the hack's behavior in a multi-process system.

20. *For the forwarding system (not screend!) he knows that one packet in should = one packet out. Thus, livelock = many packets that come in do not make it out (occassionally you may lose things even not under livelock). So the key is to figure out roughly what the MLFRR is and then never exceed that packet input rate. A simple approach is to gradually increase the packet delivery rate until he sees a non-negligible loss rate. A more clever approach would be to do something like a binary search.*

21. *Worse workload: random writes, followed by sequential reads. It should do relatively better on a RAID since two disks could be seeking in parallel. You could also state that it wouldn't do better since a well-organized RAID could stream large files.*

22. *Worst case is that we touch 5 segments in total: one segment with the block, second that contains the triple indirect, third that contains the double indirect, fourth that contains an indirect, fifth that contains an inode. How can they get so smeared? Key insight is that every write to meta data changes its location. So imagine we've written to the block. All five pieces will be put in the same segment. Imagine we write to an adjacent block some time later, putting the four above the initial block in this new segment. Repeat this process going up the tree to smear things around.*

    Problem this creates is that in the worst case cleaning generates more garbage that it removes.

23. *Process packets as long as they come in; the point of polling was for fairness.*

    To get full credit you needed to mention how polling with no quota specifically causes livelock (not just, "packets getting dropped at output queue").

24. (a) *GoogleFS: validity and elimination of duplicate record appends is not builtin to the file system, but is synthesized by clients.*

(b) LBFS: does not guarantee that the database is consistent but uses an end-to-end check to tell when its invalid and then regenerates it.

(c) GoogleFS: system does not take request and guarantee it gets out there, client has to sit and retry if it fails.

25. *There were many examples that were given full credit. Given full access to send arbitrary packets, you can first observe the packets the client sends to a server, then mimick these packets to perform arbitary operations on behalf of the client. Similarly, you may spoof the client by sending it responses that are bogus on behalf of the server. For example, if the client tries to run a program (e.g., netscape) that is located on the server, you could send back different executable blocks instead of the actual application. These blocks could contain the code to read/write data.*

26. *Both will shut off work generation when processing packets. The first by setting a flag and disabling interrupts, the second by doing everything in the interrupt handler, also preventing further interrupts from occuring until it is finished.*

27. *create, delete, etc. Also, file permissions may change onb open files but they should remain available. Also, according to unix semantics, files opened and subsequently deleted should remain available for reading. Clients can tag each request with a unique integer; server can track the last integers used and not do the operation if they match. Also accepted were fixes that mentioned a replay cache. If no fix was mentioned, the answer got 3 points.*