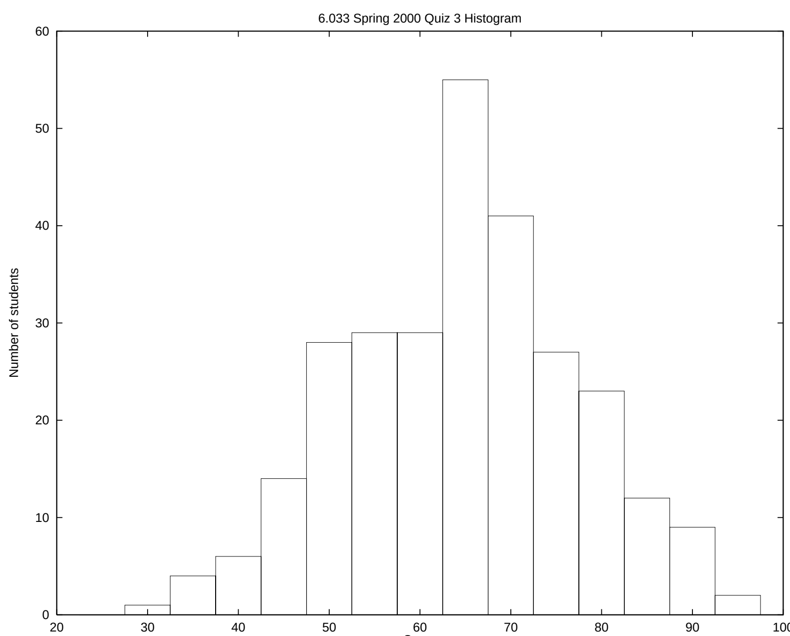
Department of Electrical Engineering and Computer Science
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033 Computer Systems Engineering: Spring 2000

Handout 41 - Quiz 3 Solutions

May 11, 2000

Average: 61.7
Median: 62
Std. Dev: 12.6



I Reading questions

1. [6 points]: Which of the following are true statements about RAID levels?
(Circle all that apply)

A. RAID 5 provides strictly better performance than RAID 3 under all workloads.

B. RAID 1 has better performance than RAID 5 for a workload consisting of small reads and small writes.

C. RAID 5 usually provides at least as good write performance as RAID 4.

D. RAID 3 has a lower disk overhead than RAID 1 when the number of disks is larger than two.

2. [6 points]: Why is the log in System R duplicated on two disks?
(Circle the best answer)

A. To tolerate software failures.

B. To tolerate hardware-decay failures.

C. To tolerate power failures.

D. To improve performance, since reading simultaneously from two disks is faster than from just one.

3. [6 points]: System R uses both shadowing and logging. According to the paper in the readings, the designers of this system are of the opinion that:
(Circle all that apply)

A. Shadowing is a bad idea that should not have been used.

B. Shadowing is a good idea for private files.

C. Logging is better than shadowing for large, shared files.

D. Shadowing makes it unnecessary to do logging.

4. [6 points]: Consider a log-structured file system (LFS) as implemented in the Sprite operating system. Which of these is true?

(Circle all that apply)

A. Data files belonging to different directories can never be in the same LFS segment.

B. Immediately after the segment cleaning procedure completes, there may be more than one partially-written segment.

C. Any segment can only contain data that were written to disk between two consecutive operations of the segment cleaner.

D. Crash recovery is fast because the latest checkpoint region is a consistent snapshot of the system; reading the last checkpoint region to restore state is the only operation that is done during recovery.

Although the paper spends a significant amount of space describing a roll-forward procedure for recovering changes after the last checkpoint, in the end it reports that roll-forward was not actually implemented; to compensate for this omission, the implemented system takes more frequent checkpoints than one might expect.

5. [6 points]: Which of the following statements is true in the Coda file system?

(Circle all that apply)

A. Only files specified in the hoard profile can be manipulated while disconnected.

B. Coda uses a *lease* to place a time bound on how long a disconnected client can write to a file before it needs to be reintegrated.

C. *All* file system operations performed during the emulation phase must be replayed in the reintegration phase for correctness.

D. Although there is little write-sharing in practice, the reintegration phase is unable to resolve all conflicts automatically.

6. [6 points]: According to Saltzer, what are some ways of coping with complexity in systems?

(Circle all that apply)

A. Feedback loops to detect errors early.

B. A single, dictatorial designer in charge.

C. Studying failed systems to learn why they failed.

D. Taking 6.033.

7. [6 points]: According to Crawley, the difference between a "complex system" and a "complicated system" is:

(Circle all that apply)

A. All in one's mind; they are simply two ways to describe the same thing.

B. Similar to the difference between "stress" and "strain" in physical systems.

C. The difference between a large number of interconnections and human ability to comprehend the interconnections.

D. Something for you to discuss with the person sitting next to you in lecture.

Stress is the force placed on a mechanical component; strain is the distance that the component distorts.

II ColdFusion

Alyssa P. Hacker and Ben Bitdiddle are designing a hot new system, called *ColdFusion*, whose goal is to allow users to backup the contents of their computers on to a distributed network of ColdFusion servers.

Users interact with ColdFusion using PUT and GET operations. PUT(data, fid) takes a data buffer and reliably stores it under a unique identifier fid, a positive integer, on some subset of the servers. It returns SUCCESS if it was successful in storing it on all the machines in the chosen subset, and FAILURE otherwise. GET(fid) returns the contents of the most recent successful PUT to the system for the file identified by fid.

Because high availability is a key competitive advantage, Alyssa decides to replicate user data on more than one server machine. But rather than replicate each file on *every* server, she decides to be clever and use only a subset of the servers for each file. Thus, the PUT of a file stores it on some number (A) of the servers, invoking a server_put operation on each server. server_put is *atomic* and is implemented by each server. If PUT is unable to successfully store the file on A servers, it returns FAILURE.

When a client does a GET of the file, the GET software attempts to retrieve the file from some subset of the servers and picks the version using an election algorithm. It chooses B servers to read data from, using an *atomic* server_get operation implemented by each server, following which it calls pick_majority(). pick_majority returns valid data corresponding to a version that is shared by *more than 50%* of the B copies retrieved, and NULL otherwise. Even though the client may not know which specific servers hold the current copy, the number of servers (A) in PUT and the number (B) in GET are chosen so that if a client GET succeeds, it is assured of the most-recent copy.

They write the following code for PUT and GET. There are S servers in all, and $S > 2$. The particular ordering of the servers in the code below may be different at different clients, but all clients have the same list. They hire you as a consultant to help them figure out the missing parameters (A and B) and analyze the system for correctness.

```
/* IP addresses of the servers; S (>2) in all. */
ip_addr server[S];    /* each client may have a different ordering */
int PUT(char *data, int fid)
{
    int ntried, nputs = 0;       /* # of servers tried and # successfully put */

    for (ntried = 0; ntried < S; ntried = ntried + 1) {
        /* Put ``data'' into a file identified by ``fid'' at server[ntried] */
        status = server_put(data, fid, server[ntried]);
        if (status == SUCCESS)
            nputs = nputs + 1;
        if (nputs >= A)          /* yes!  have server_put() to A servers! */
            return SUCCESS;
    }
    return FAILURE;              /* found < A servers to server_put() to */
}
```

```
int GET(int fid, char *data)
{
    int ntried, ngets = 0; /* # tried and # for which server_get worked */
    char files[S][MAX_FILE_LENGTH]; /* array of files; each element is
                                       for a copy from each server */
    int index;
    char *data = NULL;

    for (ntried = 0; ntried < S; ntried = ntried + 1) {
        /* Get file ``fid'' into buffer files[ntried] from server[ntried] */
        status = server_get(fid, files[ntried], server[ntried]);
        if (status == SUCCESS)
            ngets = ngets + 1;
        if (ngets >= B) {    /* yes!  have gotten data from B servers */
            /*
             * pick_majority() takes the array of files and magically
             * knows which ones are valid.  It scans the ``ngets'' valid
             * ones and returns an index in the files[] array for one
             * of the good copies, which corresponds to a version returned
             * by more than 50% of the servers.
             * Otherwise, it returns -1.
             * If ngets == 1, pick_majority() simply returns an index to
             * that version.
             */
            index = pick_majority(files, ngets);
            if (index != -1)
                copy(data, files[index]);  /* copy into ``data'' buffer */
                return SUCCESS;
            } else
                return FAILURE;
        }
    }
    return FAILURE;  /* didn't find B servers to server_get from */
}
```

For questions 8 through 11 (starting on the next page), assume that operations execute serially (i.e., there is no concurrency). You may also assume that the end-to-end protocol correctly handles all packet losses and delivers messages in the order in which they were sent. No operations are prevented from completing because of lost or reordered packets. However, servers may crash and subsequently recover.

8. [6 points]: What reliability technique studied in 6.033 comes closest to the one being attempted by Alyssa and Ben?

(Circle the best answer)

A. Fail-safe design.

B. N-modular redundancy.

C. Pair-and-compare.

D. Temporal redundancy.

9. [8 points]: Which of the following combinations of A and B in the code above ensures that GET returns the results of the last successful PUT, as long as no servers fail? (Here, $\lfloor x \rfloor$ is the largest integer $\leq x$, and $\lceil x \rceil$ is the smallest integer $\geq x$. Remember also that $S > 2$.)

(Circle all that apply)

A. $A = 1, B = S$.

B. $A = \lceil S/3 \rceil, B = S$.

C. $A = \lceil S/2 \rceil, B = S$.

D. $A = \lceil 3S/4 \rceil, B = \lfloor S/2 \rfloor + 1$.

E. $A = S, B = 1$.

Choice **C** *does not work when S is even.*

10. [6 points]: Suppose that the number of servers S is an odd number larger than 2, and that the number of servers, A, used for PUT, is $\lceil S/2 \rceil$. If only PUT and no GET operations are done, how does the MTTF of the PUT operation change as S increases? The PUT operation fails if the return value from PUT is FAILURE. Assume that the process that causes servers to fail is *memoryless*, and that no repairs are done.

(Circle the best answer)

A. As S increases, there is more redundancy. So, the MTTF *increases.*

B. As S increases, one still needs about one-half of the servers to be accessible for a successful PUT. So, the MTTF *does not change* with S.

C. As S increases the MTTF *decreases* even though we have more servers in the system.

D. The MTTF is *not a monotonic function* of S; it first decreases and then increases.

The failure characteristics of PUT *are congruent to those of the multi-engine airplane described in chapter 7. The* PUT *MTTF coefficient when there are $2k-1$ servers is $1/(2k-1)+1/(2k-2)+\ldots+1/k$. When there are $2k+1$ servers, the MTTF coefficient is $1/(2k+1)+1/2k+\ldots+1/(k+1), k \geq 1$.The difference between the two is equal to $1/k - \{1/2k + 1/(2k+1)\} = 1/2k - 1/(2k+1) > 0$. Hence, the* PUT *MTTF decreases as the number of servers increases, if we have an odd number of servers.*

11. [6 points]: Which of the following is true of ColdFusion's PUT and GET operations, for choices of A and B that guarantee that GET successfully returns the data from the last successful PUT when no servers fail.

(Circle the best answer)

A. A PUT that fails because some server was unavailable to it, done after a successful PUT, *may* cause subsequent GET attempts to fail, even if B servers are available.

B. A failed PUT attempt done after a successful PUT *cannot* cause subsequent GET attempts to fail if B servers are available.

C. A failed PUT attempt done after a successful PUT *always* causes subsequent GET attempts to fail, even if B servers are available.

D. None of the above.

ColdFusion unveils their system for use on the Internet with $S = 15$ servers, using $A = 2S/3$ and $B = 2S/3+1$. However, they find that the specifications are not always met—several times, GET does not return the data from the last PUT that returned SUCCESS.

You must allow for the possibility of concurrent operations in the rest of the questions on this quiz. S, A, and B are as specified above.

12. [8 points]: Under which of these scenarios does ColdFusion *always* meet its specification (i.e., GET returns SUCCESS *and* the data corresponding to the last successful PUT)?

(Circle all that apply)

A. There is no scenario under which ColdFusion meets its specification for this choice of A and B.

B. When a user PUTs data to a file with some fid, and at about the same time someone else PUTs different data to the same fid.

C. When a user PUTs a file successfully from her computer at home, drives to work and attempts to GET the file an hour later. In the meantime, no one performs any PUT operations to the same file, but three of the servers crash and are unavailable when she does her GET.

D. When the PUT of a file succeeds at some point in time, but some subsequent PUTs fail because some servers are unavailable, and then a GET is done to that file, which returns SUCCESS.

You tell Ben to pay attention to multi-site coordination, and he implements his version of the two-phase commit protocol. Here, each server maintains a log containing READY (a new record he has invented), ABORT, and COMMIT records. The server always returns the last COMMITed version of a file to a client.

In Ben's protocol, when the client PUTs a file, the server returns SUCCESS or FAILURE as before. If it returns SUCCESS, the server appends a READY entry for this fid in its log. If the client sees that all the servers it asked returns SUCCESS, it sends a message asking them all to COMMIT. When a server receives this message, it writes a COMMIT entry in its log together with the file's fid. On the other hand, if even one of the servers returns FAILURE, the client sends a message to all the servers asking them to abort the operation, and each server writes an ABORT entry in its log. Finally, if a server gets a server_put request for some fid that is in the READY state, it returns FAILURE to the requesting client.

13. [8 points]: Under which of these scenarios does ColdFusion *always* meet its specification (i.e., GET returns SUCCESS *and* the data corresponding to the last successful PUT)?
(Circle all that apply)

A. There is no scenario under which ColdFusion meets its specification for this choice of A and B.

B. When a user PUTs data to a file with some fid, and at about the same time someone else PUTs different data to the same fid.

C. When a user PUTs a file successfully from her computer at home, drives to work and attempts to GET the file an hour later. In the meantime, no one performs any PUT operations to the same file, but three of the servers crash and are unavailable when she does her GET.

D. When the PUT of a file succeeds at some point in time and the corresponding COMMIT messages have reached the servers, but some subsequent PUTs fail because some servers are unavailable, and then a GET is done to that file, which returns SUCCESS.

14. [6 points]: When a server crashes and recovers, the original clients that initiated PUTs may be unreachable. This makes it hard for a server to know the status of its READY actions, since it cannot ask the clients that originated them. Assuming that at no more than one server is unavailable at any time in the system, which of the following strategies allows a server to correctly learn the status of a past READY action when it recovers from a crash?
(Circle all that apply)

A. Contact any server that is up and running and call server_get with the file's fid; if the server responds, change READY to COMMIT in the log.

B. Ask all the other servers that are up and running using server_get() with the file's fid; if more than 50% of the other servers respond with identical data, just change READY to COMMIT.

C. Pretend to be a client and invoke GET with the file's fid; if GET is successful and the data returned is the same as what is at this server, just change READY to COMMIT.

D. None of the above.

Choice **B** *is almost correct, but not quite. The recovering server also needs to check that the data obtained for the file is the same as the locally stored version in* READY *state. Choice* **C** *does this correctly.*

To accommodate the possibility of users operating on entire directories at once, ColdFusion adds a two-phase locking protocol on individual files within a directory. Alyssa and Ben find that although this sometimes works, deadlocks do occur when a GET owns some locks that a PUT needs, and vice versa.

15. [8 points]: Ben analyzes the problem and comes up with several "solutions" (as usual). Which of his proposals will actually work, *always* preventing deadlocks from happening?

(Circle all that apply)

A. Ensure that the actions grab locks for individual files in increasing order of the fid of the file.

B. Ensure that no two actions grab locks for individual files in the same order.

C. Assign an incrementing timestamp to each action when it starts. If action A_i needs a lock owned by action A_j with a *larger* timestamp, abort action A_i and continue.

D. Assign an incrementing timestamp to each action when it starts. If action A_i needs a lock owned by action A_j with a *smaller* timestamp, abort action A_i and continue.

16. [2 points]: To make sure that 6.033 and this long term haven't sapped your general knowledge of other material, tell us what you think the term "cold fusion" refers to:

(Circle all that apply)

A. The way in which the sun produces its energy.

B. A claim that nuclear fusion, producing usable amounts of energy, can be induced to take place on a table-top by electrolyzing heavy water with expensive electrodes.

C. The technology behind the World War II atomic bombs.

D. For the last time—there's a reason I'm in Course VI, not Course VIII or Course XXII!

E. Shouldn't we be at the end of 6.033 by now?

Have a great summer!!!