739 Midterm: Fall ‘17

Sometimes, it’s just a bunch of questions

Usually, exams have something fun that ties them together. But not always. In this case, there is one thing that unites this exam: it’s a bunch of questions about distributed systems, some small, some bigger, and hopefully, if we’re lucky, unusual enough to take you somewhere new in your thoughts. Think, then answer. Good luck!

 NAME: ____________________

1. Hamilton, in his advice paper on internet services, writes the following:
   [You should have] "zero trust of underlying components". What does he mean? Is this sensible advice?

Lots of possible answers here. Zero trust means components will break, systems will crash, and in general stuff doesn't work well all the time, so you shouldn't rely on such "good" behavior.

In that sense, it is good advice.

However, it is also somewhat overstated. Clearly, there is some trust. Disks, for example, are expected generally to try to keep some data, CPU instructions generally work, etc. Thus, place some trust, but think carefully about what happens when components fail, and make sure you can handle that.

2. Dean advocates the use of "back of the envelope" calculations.
   (a) Why is this useful for system architects?
   (b) In WiscKey, you read about LSMs and some of their advantages. Perform a back-of-the-envelope calculation that demonstrates how much write amplification is tolerable in LSMs on hard drives (i.e., how many sequential writes can one do to replace some random ones?)

Back-of-the-envelope is useful for system design because you can quickly tell which designs are going to work well and which are not. To do this well, you need to know basic costs of system operations (e.g., cache miss, branch mispredict, disk I/O, etc.) as Dean articulates.

As for LSMs, the basic idea is to transform all writes into sequential ones, and thus improve performance. Typically, hard drives are roughly 100x-1000x faster doing sequential I/O (e.g., 100MB/s vs 1MB/s). Thus, if one random write can be replaced by N sequential I/Os, it's ok, as long as N < 100 (or so). With LSMs, data does get rewritten many times due to compaction across levels; as long as the amplification level stays lower than the difference in cost, the tradeoff should be worthwhile.

3. In the RPC paper, the server receives requests from clients and sets up per "activity" information to track this interaction.
   (a) What is an activity?
   (b) How does the server know when it can garbage collect an activity and related info?
   (c) Finally, what could go wrong if the server prematurely garbage collects such info?

a) activity is (machine ID, process ID) pair. It uniquely identifies a communication from a particular client

b) the server tracks info on a per-activity basis. If there is no communication for some period of time from a client, the server can GC that info. The "period" is determined by the time after which the server is not in danger of receiving a retransmitted message (e.g., 5 minutes)

c) the server would then possibly not deliver "at most once" semantics. A call could be serviced, and then the reply lost. If the server then GCs the info, a retransmit from the client could cause the RPC to be executed a second time on the server, as this now would look like a new request.

4. Gray shows how to calculate availability as a function of the mean-time-between-failure and the mean-time-to-repair.
   (a) What is this equation? (i.e., Availability = …?)
   (b) Assuming we want a system that has 99.99% uptime and has a mean-time-between-failure of 1 day; how fast does the system have to repair itself in order to reach this availability goal? (approximately)

a) Avail = MTBF / (MTBF + MTTR)

b) 0.9999 = 1 day / (1 day + MTTR)
   0.9999 * (1 day + MTTR) = 1 day
   0.9999 MTTR = .0001 day
   MTTR = .0001 day / 0.9999 = (1/9999) days = ~9 seconds

5. In the Toronto paper about bugs, the following code is shown to both indicate a bug and possible fix:

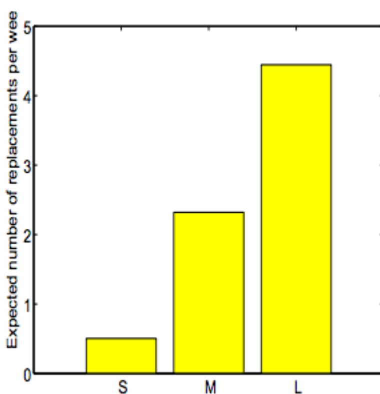
```
try {
    split(…);
} catch {Exception e) {
    LOG.error("split failed…");
+   retry_split(); // fixed!
}
```

(a) Why was this a bug without the added line?
(b) What is potentially bad about this solution?

a) Because it just logs that something bad happened; it doesn't do anything about it!

b) The solution retries immediately — if the system keeps failing, infinite retry (without backoff) can be a bad solution (and can even cause denial of service)

6. The Schroeder paper on disk failure makes a number of surprising conclusions. From Figure 7 (shown here), they show the number of disk replacements in a given week, bucketed by the number of replacements in the previous week. What can you conclude from this graph, and why is it important?



In systems, we want failures to be independent along all axes. This graph shows some dependence, which can be worrying.

Specifically, it shows that in a week with a small number of failures, the number of failures that will occur in the following week will be small; medium means medium, and large means large. Thus, there are correlations in failures which can make designing reliable systems challenging.

7. In the study of Flash failures from CMU, the authors state "The amount of data written by the operating system to an SSD is not the same as the amount of data that is eventually written to flash cells." Why do they state this, and why might it be important?

They state this because you might be tempted to measure I/O from the OS and conclude that this is the exact amount of I/O the drive performs to its chips; if you do this, you might get a bad estimate (for example) of when the drive will wear out. Instead, if possible, you should get the actual count, not some approximation from higher up in the storage stack.

8. Lamport's clock condition states: For any events a, b: if $a \rightarrow b$, then $C(a) < C(b)$.
   (a) Given two logical clock values for processes on the same machine, what can we conclude about the ordering of the two events?
   (b) What if the clock values we wish to compare were from events on different machines?

a) assuming they are in the same process, or share a clock, you can know that one event (with the lower clock value) preceded the other (with the higher value).

b) nothing, really. imagine the case where two machines have not communicated yet, there is no way to say anything given values from each machine.

9. Vector clocks use more information than Lamport clocks. Give an example where vectors clocks are more useful than Lamport clocks.

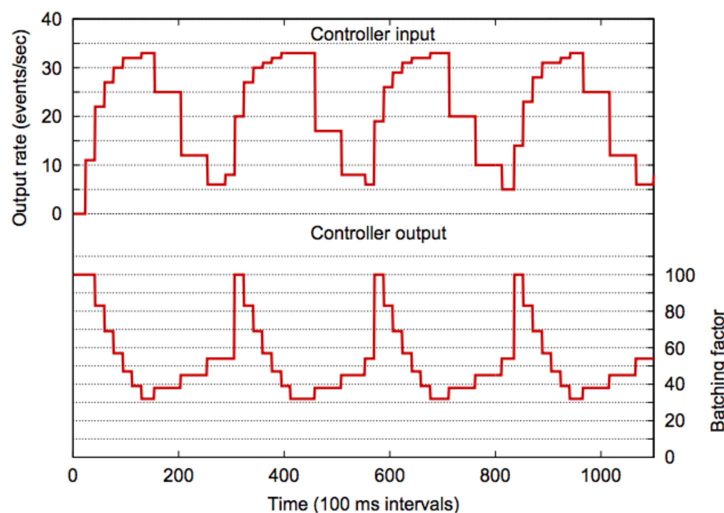
Vector clocks are more useful when you wish to be able to reliably determine whether events are causal or concurrent. With Lamport, you cannot always tell this because information is lost; vector clocks preserve more info and thus can always be compared to determine causality or concurrence.

10. The Flash web server paper introduces the single-process event-driven (SPED) architecture. What is SPED, and what are the fundamental limits of such an approach?

SPED is a single process event driven approach. The idea is to use event-based processing because it is fast and gives full control over scheduling.

Some limits to this approach are (1) it has problems with synchronous I/O and (2) it has problems scaling to use multiple CPUs. (1) is addressed readily with helper processes that issue the I/Os. (2) can be addressed, but introduces all the same sort of synchronization problems previously avoided in event-based approaches.

11. SEDA implements a number of "controllers" to help with performance. One such controller is the "batching" controller, as shown in this graph. What is the batching controller doing, and why is it useful? (what tradeoff is it managing?)



The batching controller controls how many events are queued up before a handler is given the events to process.

Waiting too long increases latency; not waiting long enough increases overhead as little grouping of events takes place and thus the handler must be repeatedly called.

Adjusting it dynamically can tune for the sweet spot of good bandwidth and not-too-high latency.

12. The ALICE paper explores how file systems implement crash consistency and how applications built atop such file systems implement crash consistency as well. Why is crash consistency implemented at both of these levels of the system? (why not just in the application, or just in the file system?)

Crash consistency must be implemented in the file system for (minimally) file system metadata. No application can implement crash consistency for the file system metadata, because it is not generally visible at that layer.

Applications must implement crash consistency for their data because the guarantees provided by the file system are not generally good enough. For example, an application may wish to commit a multi-file atomic update; because the file system has no such interface, the application must use some kind of protocol (e.g., write-ahead logging) to realize such a goal.

13. The WiscKey paper shows that both read and write amplification are a problem in LSMs (especially in classic LevelDB).
    (a) Describe what these two amplifications are.
    (b) In LevelDB, which is worse and why?
    (c) Finally, in what ways does WiscKey reduce read/write amplification?

a) read amplification: the total amount of I/O done over the amount of I/O requested for reading; write is the same but for writes

b) Fig 2 shows that read amplification is worse than write amplification. It is worse because lookup is generally harder; not only do you have to consult each level of the tree, but once you find the right file, you have to perform many reads to finally find the correct data.

c) WiscKey's major reduction of amplification occurs because it separates key from values. Values are written once to a log. Thus when compaction is occurring during database loading, the values do not have to be repeatedly read/written during compaction (only the keys are). Thus, total I/O due to compaction is greatly reduced.

14. The NFS system utilizes idempotency where possible to allow retry of operations in case of failure with little ill effect. However, some operations are not (generally) idempotent.
    (a) An example is "mkdir"; why is mkdir not idempotent?
    (b) Could you design a mkdir with different semantics that is idempotent?

a) mkdir is not idempotent because repeating it does not get the same results as doing it once, if the directory does not already exist. Specifically, the first time a new dir is created the operation succeeds, but after that it fails (saying "dir already exists")

b) sure, you could. many examples possible. one: mkdir does not fail if the directory already exists, but only if it truly cannot create the directory at all.

15. Assume the following protocol used in a distributed storage system: to write block X, a client acquires a lease for X (i.e., lease(X)) from a lease server; it then writes to shared storage at location X; it then contacts the lease server to release lease(X). Other clients follow the same protocol. Is this protocol correct? What can go wrong?

The protocol is error prone, because a client could grab the lease, get stuck doing something else for a while (e.g., swapping to disk, GC'ing), and then do the write. However, at that point, the lease has already expired, and thus the write should not be allowed, because another client may have a valid lease. More is needed here to make this work, like some sort of "write fence" to prevent the late write from taking place.

16. We now discuss HA-NFS.
    (a) How does HA-NFS handle disk, server, and network failures?
    (b) Overall, what do you think of the HA-NFS design? (its strengths, its weaknesses?)

a) disk failure via local mirroring on each server; servers run as primary/backup for each other, and thus can detect and failover as need be; networks are replicated and thus can handle some amount of failure too.

b) its strength is that it takes NFS and makes it highly available without too much added machinery. a weakness is that split-brain can still occur (despite precaution), and that each server can only run at half loads (max) if we want to be able to truly handle the load from a failed server on one machine when the failure occurs.

17. Bonus: We saw two talks in class: the first was from Muthian Sivathanu (MSR) talking about his experiences inside Google; the second was from Venkat Venkataramani (RockSet) talking about his new work at a startup. Describe one thing you learned from each talk.

Lots possible here.

From Muthian: flash can be faster than memory(!), if done right!

From Venkat: cloud-native is the new way to design; you have as many resources as you need, so use them!