Getting Advice
(CS739 Fall ‘16 Midterm)

“No enemy is worse than bad advice” —Sophocles

The papers we read are forms of **advice**: lessons on what to do, and what not to do, when building or measuring systems. Your job on this exam: examine each piece of advice and decide whether it is good or bad advice, as well as a few other things. Read each question carefully and answer the best you can.

Good luck! And remember this OS-inspired advice: answer questions in **EASIEST-QUESTION-FIRST (EQF)** discipline. Following this advice will maximize the number of questions you finish before your time is up.

Name: R. U. Havinfun

Student ID: 123 456 7890

1. A person named J. Hamilton gives a lot of advice in his paper “On Designing and Deploying Internet Services.” One piece of advice he gives is this: “Support a big red switch.” What is the big red switch, and is having one in your system good advice or bad?

From the paper: “The concept of a big red switch is to keep the vital processing progressing while shedding or delaying some non-critical workload.”

Answers on whether this is good or bad were both accepted, depending on the reasoning.

2. Jeff Dean gives a lot of advice about how to build systems, including numbers “everyone should know” (such as cache miss latency, mutex lock/unlock time, etc.) Why does he give the advice that everyone should know these numbers? Why are such numbers useful to system builders?

From the slides: “Back of the envelope helps identify most promising [design variation]”

Most good answers focused on this aid to designers as the key thing.

3. The RPC paper gives advice, of a sort, to send large packets by sending a smaller piece, waiting for an acknowledgement, then the next smaller piece, then waiting for an ack, etc. Is this good advice? When does this advice make sense, and when doesn’t it?

Sending a large packet one piece at a time is slow, increasing latency and lowering bandwidth. The authors do this because it is (a) easier to implement and (b) they think their workload doesn’t need to worry about large packets. The advice makes sense if (b) holds true.

4. U-Net advocates (strongly) a position that networking be moved (mostly) to user level, yet despite this advice, some pieces of the U-net system remain in the kernel. Which pieces of networking functionality are placed in the kernel with U-net, and why?

Key parts relate to security and some other setup (non data path) tasks. From the paper: " The operating system service will assist in route discovery, switch-path setup and other (signalling) tasks that are specific for the network technology used."

The kernel can also multiplex many (slower) connections onto a single physical one, but this is not the main point and clearly doesn't help with performance (but rather is for completeness of functionality).

5. Gray says the key to high availability (HA) is to "modularize the system". Why does he give us this advice, and what does it mean? Can you have HA without such modularization?

As the paper states: "VonNeumann's model had redundancy without modularity. In contrast, modern computer systems are constructed in a modular fashion a failure within a module only affects that module. In addition each module is constructed to be fail-fast -- the module either functions properly or stops. Combining redundancy with modularity allows one to use a redundancy of two rather than 20,000."

So yes, you can have HA without it, but at high cost.

6. The Ding paper on "simple testing" shows the following diagram. What advice would you give system designers, based on this diagram? (put most important advice first)

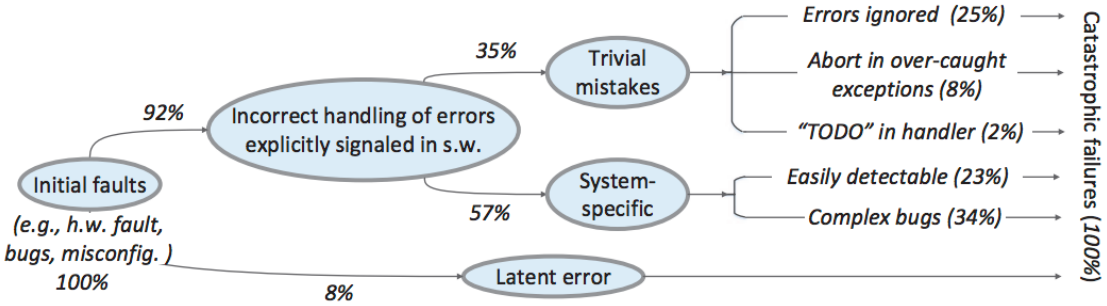

Figure 5: Break-down of all catastrophic failures by their error handling.

- ***Unit test everything.***
- ***Check error handling carefully and implement it throughout.***
- ***Use tools where possible to find other simple problems (e.g., TODO that isn't DONE).***
- ***etc.***

7. Schroeder concludes, in her study of disk failure, that disk failures do not take on a bathtub-like curve. What is a better depiction of the failure-rate curve she discovers? (draw this) Given this new curve, what advice would you give to large-scale storage system designers?

Graph misses the first part of the bathtub (and to some extent the middle) and goes up.

Advice could be in many forms. One example:

- ***Expect failures earlier and at an increasing rate with age***
- ***Perhaps consider replication strategies across drives that take age into account***
- ***Perhaps consider data migration strategies so as to minimize vulnerability in case of old drive failure***

8. The ALICE paper discusses application-level protocols, and gives this piece of advice for Linux: when needed, make sure to call fsync() on the file of interest as well as on the parent directory. Why is this advice necessary? Does one need to always follow this advice to build a correct data-management application (such as the ones mentioned in the paper)?

From the paper: "An fsync() on a file does not guarantee that the file's directory entry is also persisted."

May lose a file because directory isn't flushed with crash at inopportune moment.

Always necessary? No. If your protocol doesn't create new files, no problem. Also, if you only run on file systems that persist the directory entry on fsync() you are also OK.

9. In Sun's network file system, the basic protocol works particularly well for idempotent operations, so perhaps the advice here is to "always make operations idempotent". Despite this advice, not all NFS operations are idempotent. What types of operations are idempotent, and which are not? What kind of problems arise when non-idempotent operations are executed?

Good answers talked about mkdir() or create() or related protocol parts that, when repeated, lead to a different answer than when just executed once (e.g., re-creating a file that exists leads to an error).

The problem is thus user confusion.

Solution (which some provided but not strictly needed) is a replay cache of some kind as discussed in the HA NFS paper.

10. The two-phase commit algorithm we discussed in class advises that some logging steps need not be forced to disk immediately. Describe at least one such case, and explain why it is a good idea to follow this advice.

Many possibilities. From paper, the PrN protocol coordinator: "To commit a transaction, a PrN coordinator does two log writes, the commit record (forced) and the transaction end record (not forced)." Worst case here is a crash that has the coordinator ask cohorts for ACKs, which is no big deal because it is rare.

It's a good idea for performance; non-forced means you can delay and batch writes to disk. Forced means you have to wait and that is slow.

11. Remus describes an approach to high availability using "speculation". They say that this form of speculation is needed to make their VMM-based HA system operate well. Why do they give this advice? First, describe what they mean by speculation, and then describe why it makes Remus work well.

Remus works by having the primary work on many requests concurrently and only take snapshots occasionally. While the primary works on these requests, it avoids externalizing their results. Then, Remus takes a snapshot of the system (memory + disk state) and ships it to the backup, at which point it is safely replicated in backup memory and thus the primary can release outputs and allow clients to see the results of their requests.

This does NOT improve latency; in fact, it increases it! But it does improve throughput as compared to an approach that does one request at a time. Thus, Remus's approach allows the batching of requests and increase of throughput.

12. The WiscKey paper advocates for the separation of keys and values in an LSM-based key-value store. When is this a good piece of advice, and when is it bad?

Good answers focused on the paper and its main results. WiscKey works well under write workloads (pretty much all), as well as most read workloads where value sizes are large. WiscKey has trouble under range read workloads where the data was inserted in random order.

Less "good" answers focused on what would happen when you run on hard drives, which is just speculation.

13. The Flash paper advises the use of “helper processes” in some cases. When are such processes needed? Is it ever a bad idea to use helper processes?

Some operations, such as a synchronous disk I/O, “block” and thus stop a process from running until an I/O is complete. A helper process (or pool of processes) can help avoid blocking in an event-driven server by taking on such requests itself, and then later notifying the main process when the request has completed. Not blocking is critical for event based systems; if the main event loop blocks, NOTHING gets done until the blocking is complete.

It can be a bad idea for a number of reasons. For example, if asynchronous I/O APIs exist, the overhead of moving data back and forth between main event thread and the backup processes might be (much) more costly.

14. In both papers on latent-sector errors and disk corruptions, Bairavasundaram et al. note that such failures are **not** independent. What advice would you give designers of RAID storage arrays (or other replicated storage systems) assuming this is indeed the case?

To types of non-independence of these localized failures: spatial and temporal. Spatial may be easier to deal with: for example, be careful of placing important data near other important data within the same disk (e.g., a block and its parent); you could also extend this across disks to deal with weird cases.

Temporal is harder to deal with: what should a designer do to react to faults that happen close in time on one disk? Not sure if there is a good answer, but it might be a good idea to quickly react when one such fault is detected, so as to prevent further damage. If a single fault is a good predictor of soon upcoming faults, removing the bad disk and then repairing data to a new one might be a good idea.

15. A wise old systems person has been overheard giving the following piece of advice: “Always use vector clocks instead of Lamport clocks - they are strictly better!” Show a case where they indeed are better, and then decide if one should always follow this advice (and explain why).

Vector clocks can preserve more information than Lamport clocks and thus in some cases are more useful (look to the VC paper for examples). However, they can be costly, as per-node state does not scale well. Thus, if you can avoid using a full VC, you likely should.

16. In HA-NFS, the authors advise the reader to use different approaches in handling server, disk, and network failure. What was their specific advice on failure handling for these different types of failure? What do you think of this approach?

Server failure: Dual port the disks and use heartbeats for failure detection. If one server fails, the other can take over and serve its requests.

Disk failure: Optionally use disk mirroring (note: dual porting is not the solution here!)

Network failure: Redundancy of network interfaces and links to the servers are useful here.

Overall, pick each resource and provide a different type of redundancy for it, based on what is most appropriate - an interesting strategy. Most people liked it. Some didn't, usually for reasons that were unclear (too costly?).

17. The Flash failure paper from CMU by Meza et al. suggests that "throttling" may help decrease SSD failure rates. What is throttling, and why might it be useful?

From the paper: "One hypothesis is that temperature-sensitive SSDs with increasing error rates, such as Platforms A and B, may not employ as aggressive temperature reduction techniques as other platforms. While we cannot directly measure the actions the SSD controllers take in response to temperature events, we examined an event that can be correlated with temperature reduction: whether or not an SSD has throttled its operation in order to reduce its power consumption."

That was the basic idea.