REVIEW OF FAULT-TOLERANCE (CHAPTER 8)


2. A bit more on channel coding (Hamming codes)

   --Last time, we saw a simplified version of Reed-Solomon codes. That code is an _erasure-recovery_ code: it recovers from dropped frames. The reverse of this sheet tries to give a little bit more motivation than we saw last time.

   --Now we are going to quickly see an _error-recovery_ code, the Hamming code. Note that error-recovery is more powerful than erasure-recovery: an often unstated assumption of erasure-recovery codes is that a bit error corrupted a frame, making it unusable. With error-recovery, the code doesn’t have to throw away a corrupted frame: the code actually _corrects_ the bit errors.

   --Coding is a broad and deep area of research (in both EE and CS). It comes up in 6.033 as an example of how redundancy (in this case, redundant information) is useful for recovering from failures (in this case, network failures like packet drops and bit flips).

LOG-STRUCTURED FILE SYSTEM

3. Overview

4. Performance: for now, assume LFS has a disk of infinite size and that the computer never crashes. Questions:

   A. What’s going on in each of the five cases in Figure 9?
   B. Do those cases make sense?

5. Crash-recovery: we still assume LFS has a disk of infinite size but now let’s acknowledge that crashes can happen. Questions:

   A. If a crash happens, how can LFS, after booting, find the most recent version of a given inode? One option is to scan the entire log to rebuild the inode map.
   B. What’s wrong with that option?
   C. What can LFS do instead?
   D. How does doing that thing that help? What does LFS do on boot now?

6. Garbage-collection: now let’s acknowledge that the disk isn’t infinite. Questions:

   A. LFS has to garbage-collect, right? What do they call that process?
   B. How does LFS avoid garbage-collecting the whole disk at once?
   C. How, in principle, can LFS get the information about which disk blocks are "garbage" and which are not?
   D. When LFS garbage collects an area of the disk, what does it do?
   E. How does LFS decide which areas of the disk to garbage collect?

Note: concluding thoughts on reverse
CONCLUDING THOUGHTS

More on Reed-Solomon Codes

You may be wondering, "Why does the sender go to the trouble of encoding its message? Why doesn't it just send lots of copies of its message?" The answer is efficiency. The idea that went unarticulated in the last recitation (apologies) is that the sender and receiver have a belief about the probability that a drop can happen. However, they do not know *which* packets (or frames or messages) will be dropped in the network. Let's take an example with made-up numbers. The sender has 300 messages to send, but if the sender simply sent them, some would be dropped in the network. However, the sender and receiver are pretty sure that if the sender sends 400 messages, then at least 300 will arrive (in that case, they believe that the drop probability is less than 1/4), so for good measure, if the sender has 300 messages to send, it actually sends, say, 600. We are pretty sure that 75% of 600 = 450 messages will arrive, but the sender only needs for 300 to arrive. (Disclaimer: that was a simplification.)

Now if the sender simply sends each message twice, it is _extremely_ probable that for at least one message, both copies will be dropped, defeating the reliability goal.

The problem that Reed-Solomon codes are solving is how to allow the sender to send 600 messages in such a way that if *any* 300 arrive, the receiver can reconstruct the data. To tie the example above back to the last recitation, k = 300, n = 600, and the channel coding scheme described in that recitation would be able to tolerate up to (n - k) = 300 packet (or frame or message) drops. To repeat, the cool part of all of this is that it doesn’t matter *which* (n - k) messages are dropped. Note that the choice of k and n is an engineering decision, decided before deployment.

Logging

Logging is used in many different systems. As we move into the "databases" and "recovery" unit of the course (chapters 9 and 10), we will see system designers using logs for crash-recovery. In these cases, the system (e.g., a database) stores its data in a highly structured form on disk. The system also _logs_ its data. In this case, the log has nothing to do with performance; its purpose is crash-recovery.

LFS

You may be wondering, "Why aren’t people using LFS today?" We can’t be certain, but here are a few possibilities:

--performance: today’s file systems are high-enough performance that the performance benefit of a log isn’t needed.

--crash-recovery: people either don’t care about crash-recovery (look at the fact that many people use ext2, which has no ability to do crash recovery) *or* if they do care about crash-recovery, they use _aspects_ of LFS. As an example of the latter case, consider ext3: it incorporates a journal (i.e., a log) for crash-recovery but, in contrast to LFS, that log does not store all of the file system’s data.

--given the above two bullet points, there may not be much incentive for people to switch to something like LFS

--also, for particular kinds of workloads, LFS _doesn’t_ perform better (as Fig 9 shows). perhaps that fact made people uncomfortable with LFS.