COMPSCI 630 Systems **Fall 2016**

Lecture 15: Fault Tolerance, cont.

Lecturer: Emery Berger *Scribe(s): Jackson Warley*

15.1 Replication

Fault tolerance depends on replication; if data/control is lost in one part of the system, problems will ensue unless it can be restored from an auxilliary part of the system. Replication prevents a system from having a single point of failure; any system with a single point of failure is, by definition, not fault tolerant.

A topical example of a non-fault-tolerant system is the 737 MAX airplane, which caused two crashes in 2018 and 2019. These airplanes had a sensor to measure the angle of attack in order to detect a potential stall. The sensors fed into an automatic stall-prevention system that would lower the nose of the plane when the attack angle was sufficiently high. However, because the sensor erroneously reported a high attack angle, and there was no duplicate sensor to compare against, the stall-prevention system began to lower the plane automatically despite its having an appropriate angle of attack. Because there was no mechanism to transfer control from the stall-prevention device to the pilot, the continued lowering of the plane's nose eventually caused the plane to crash.

For replication to be effective, replicas should ideally be independent. In other words, for any two replica nodes N_1 and N_2, we should have

$$P(N_1 \text{ fails} \cap N_2 \text{ fails}) = P(N_1 \text{ fails})P(N_2 \text{ fails}).$$

Otherwise, replicas may be subject to *correlated failures*, where multiple nodes are likely to fail at the same time. Correlated failures greatly increase the risk of the entire system failing. For hardware, this idea applies across many aspects, including geographic location, manufacturer, and manufacturing date. For instance, servers that are all in one datacenter are likely to fail simultaneously if there's an earthquake there, and hard drives made in the same batch have been shown to have correlated failure timings. For this reason, service providers often try to ensure that backup data is stored in completely different locations from its original copies, and often use tape backups as a final layer of redundancy.

15.2 Hardware Fault Tolerance

15.2.1 Nondeterminism

A uniquely challenging problem for hardware fault tolerance is that hardware is nondeterministic. It is possible for physical bits in memory to be flipped by interactions with cosmic radiation or alpha particles (more common on Earth). These errors are relatively rare for computers operating on Earth, but they become a serious problem for memory on satelites or space vehicles. The space shuttle used "trimodular redundancy," i.e. three identical copies of the controller, which vote on each computation so that collectively they can tolerate a failure in any one controller. Other defenses involve physical shielding of the memory by dense materials, and ECC (error-correcting code) RAM, which can survive many bit flips before becoming unrecoverable.

15.2.2 RAID

RAID (redundant arrays of inexpensive disks) was developed in response to a general trend of disks becoming larger and more expensive. The core idea is to obtain high storage capacities by using an assortment of smaller disks. Usually, data that is logically contiguous is "striped" across multiple physical disks in order to increase the potential for parallelism in reading/writing.

Splitting data across multiple disks has the drawback of increasing failure rates. Assuming each disk independently has some probability $(1-p)$ of success, the probability of d disks succeeding on a given operation is $(1-p)^d$, which becomes significantly less than $(1-p)$ as more disks are added. To overcome this downside, commonly used RAID schemes use checksums so that data on a failed disk can be recovered from the other disks. However, in practice, it is unlikely that all data from a failed disk will be able to be recovered, though some of it usually can be.

RAID's desiderata of storage capacity, speed, parallelism, and fault tolerance are in some tension, and the various RAID schemes represent attempts to find points on the Pareto frontier of these axes.

15.2.3 Aside: Writing to Disk

Data is not written to a hard drive in "real time," as this would be prohibitively slow. The reason is that fragmentation often occurs, whereby data that is logically contiguous is stored in distant physical locations on the disk platter. The seek time of the drive's write head is enormously slower than any other mechanism in the entire system, so having the drive constantly write to potentially fragmented locations would be an unacceptable bottleneck. Instead, the disk controller caches blocks of data to be written all at once, so that more data can be written per seek. This is a hardware implementation detail invisible to the operating system, which means that if a fault occurs after the OS has "written," some data to disk, but before the disk controller has physically written it from the cache, the data will be lost.

15.3 Software Fault Tolerance

Fault tolerance is one of the reasons Heisenbugs may be preferable to Bohr bugs. Heisenbugs are mostly independent across instances of a piece of code, whereas Bohr bugs are fully correlated. Thus, replication provides no additional tolerance against Bohr bugs, whereas it can effectively mitigate the effects of Heisenbugs.

15.3.1 Transactions

Some usually desirable properties of transactions are:

- **Atomicity.** All of the transaction's logic occurs or fails as a single unit, unable to be interrupted by other operations.
- **Consistency.** The transaction does not induce invalid states.
- **Isolation/Integrity.** The transaction is robust against execution in a concurrent environment. Semantically equivalent sequences of transactions should have the same effects whether they are executed concurrently or serially.
- **Durability.** Upon completion, the transaction's effects should be committed to stable storage.

These desiderata are commonly abbreviated as ACID.[1] A standard implementation keeps a log of updates as each transaction is being executed. Right before the transaction completes, a message is written indicating that the transaction is about to commit. Once the transaction is committed successfully, its log data leading up to the commit can be dropped.

15.3.2 Concurrency Control

Locks are *pessimistic* concurrency control. They assume that shared mutable access to data will always cause a fault, and so shared mutable access should be categorically denied to prevent even the possibility of a concurrency bug.

There is also *optimistic* concurrency control, wherein an attempt is made to execute transactions concurrently with no locks. If another accessor actually appears during the course of a transaction, the transaction is aborted, the system is restored to a checkpoint, and another attempt is made. Otherwise, the transaction is committed. Optimistic control tends to be better when either (a) the cost of acquiring a lock is very high, or (b) there is a very low probability of conflict. Otherwise, the cost of retrying a transaction many times may outweigh the overhead cost of just using locks in the first place.

An even more aggressive extension of this idea is *failure-oblivious computing*, which allows certain (usually memory safety) errors to occur and simply continues program execution as though they had not happened. Despite making many people highly uncomfortable, this approach works better than one might expect, often transmuting fatal program errors into smaller errors that can be caught by the program's existing error-handling logic. A closely related approach is probabilistic memory safety, which also sacrifices a guarantee that no memory errors will occur, but differs from failure-oblivious computing in that it admits provable bounds on the probability of an unsafe operation.

Another related idea is *approximate computing*, which sacrifices some degree of accuracy or determinism in exchange for speed and fault-tolerance. A drastic example is *loop perforation*, which simply skips some percentage of the executions of a loop, in order to speed up a computation. Another example is Bolt, which is a program that attempts to automatically break out of infinite loops by analyzing the branches that are taken.

[1] There is also BASE, which cares more about eventual consistency.