Introduction

So far in this class, you've seen a few things:
- How operating systems enforce modularity on a single machine
- What the network in between those machines is like (what functionality it provides, etc.)

Within these environments, you've seen a few examples of how we deal with certain types of failures:
- TCP senders retransmit packets upon loss
- The Internet is a packet-switched network, which allows it to route around failed links (circuit-switched networks, like the landline phone network, can't do this). If a link in the Internet goes down, BGP (and other protocols) will find an alternate path.
- DNS allows -- in fact, mandates that -- zones to replicate nameservers. If one NS goes down, it's fine.

In the second half of this course, we're going to develop a more systematic way to deal with failures; a systematic way to provide "fault tolerance". We want our systems to keep running despite failures -- maybe even recover from such failures -- and we're going to allow a much more complicated set of failures.

Think back to the client-server model. We haven't discussed what to do, e.g., if the server fails. We've dealt with more benign errors: packet loss in the network, a user-program having a bug, etc.

Additionally, for the most part, we haven't discussed any recovery plan: once something fails, what do we do? (In the case of lost packets we have a recovery plan: we retransmit.)

As part of this, we're going to start thinking about much bigger systems than a simple client-server model. We'll imagine large, distributed systems: systems that involve interactions between multiple machines (think: hundreds, thousands), potentially located in geographically diverse environments.

To be systematic about providing fault-tolerance, we'll talk in parallel about the needs of the applications running on these systems. You'll see that the needs of these applications inform what sorts of abstractions will be useful in trying to achieve fault-tolerance.

For the next few weeks, we're going to be dealing with random
failures. After that, we'll broaden our class of failures to include malicious ones (e.g., targeted attacks).

======================================================================
Building fault-tolerant systems
======================================================================

General Approach

Here's are general approach for designing fault-tolerant systems. Much like my general approach for improving performance, it abstracts away some (all) of the difficulty.

1. Identify possible faults

   What sorts of faults?
   - Software faults: blue screens
   - Hardware faults: disk failures, cable failures, etc.
   - Design issues: Therac-25
   - Operation issues: AT&T outage
   - Environment issues: Tsunamis

2. Detect and contain

   Our systems should have mechanisms in place to detect errors and limit their propagation (note: at this point in the process, we haven't fixed any errors yet).

   Example of detection: In TCP, packets contain a checksum (think: they contain a hash of the packet). Switches on the path verify that the checksum matches at each step, drop packets if it doesn't.

   We also talked about this in BitTorrent: the .torrent file contains hashes of the pieces of the file.

   Example of containment: We used the client/server model to limit the effects of an error on the client.

3. Decide a plan for handling the fault. There are numerous options:

   - Do nothing
   - Fail-fast: detect and report the error at the next higher-level module.
   - Fail-stop: stop immediately upon detection. This contains errors, but means the higher-level module has to find a means to discover the failure.
   - Mask the error
Before we begin in earnest, I'm going to get a few disappointing things out of the way:

1. Of all the types of things that can fail -- software, hardware, operations, etc. -- we're never going to be able to get any of them to be free from faults. We will *always* have a collection of unreliable components. Our goal is to build a reliable system from them.

   This means that our guarantees will essentially be probabilistic: we can't guarantee perfect reliability, but we'll try to get arbitrarily close.

2. Reliability comes at a cost -- hardware, computational resource, disk bandwidth, etc. More reliability usually costs more. Have to decide -- as with TCP -- whether the additional reliability is worth the cost. Always a tradeoff.

   The most obvious tradeoff you will see is between reliability and complexity. Making a system reliable, even as much as we try to keep it modular, is going to increase complexity.

3. This whole thing is a difficult process, because it starts by requiring you to identify all possible faults. It's easy to miss some possibilities. Hence, we iterate.

4. Even though "software faults" are among the faults we're concerned with, fault-tolerant systems do rely on (some) code being correct. Is that realistic?

   First, recognize that most software on a computer doesn't matter. If Word crashes, it's annoying, but not critical. Whereas code in the disk controller causes the whole system to not work.

   For code that does matter, we have stringent development practices: well-defined stable specification, modeling, simulation, verification, N-version programming (tricky!), etc.

   Again, it's not fool-proof. But for our purposes, we'll assume that this mission-critical code is correct.

Quantifying Reliability

How will we know if we've made a system more reliable? We need some metric. Our general goal is to improve the availability of the system, and we can measure that:

\[ \text{MTTF} = \text{mean time to failure} \]
MTTR = mean time to repair  
MTBF = mean time between failures (MTTF + MTTR)  
availability = MTTF / MTBF

Example: Suppose my OS crashes once every month, and takes 10 minutes to recover.  

MTTF = 30 days = 720 hours = 43,200 minutes  
MTTR = 10 minutes  
MTBF = 43,210 minutes  
availability = 43,200 / 43,210 = .9997  
=> two hours of downtime per year

Reliability via Replication, specifically on Disks

To improve availability, we have one general technique: redundancy.  
Redundancy comes about in many forms:  
- Replication: replicate critical components  
- Error-correcting codes: add redundant data to detect/correct bit errors  
- Have multiple paths: if one fails, you have a backup

We're going to spend the rest of this lecture talking about replication, and specifically replication on a single machine: on disks. Tomorrow in recitation, you'll see replication across machines.

Why do disks fail?

Why talk about disks? Starting from the ground up: improve reliability as much as possible on a single machine before we branch out to multiple machines.

But why disks and not another part of a machine? Unlike other components, the disk is the one where, if it fails, your data is gone. You can always replace the CPU or the cable, or restart the file system, but replacing the disk after failure means your system won't work. So the cost of a disk failure is high.

Disks can fail in any number of ways:  
- The whole disk can stop working (electrical fault, broken wire, etc.)  
- Sector failures  
  - R/W from wrong place (e.g., if drive is bumped)  
  - Fail to R/W because head is too high, or magnetic coating is too thin
- Disk head contacts drive surface and scratches it

Are these failures frequent? Manufacturers claim that the MTTF is 700,000+ hours (80 years). 80 years ago we were programming with abacuses.

But that's unrealistic. Here's how they (likely) measured it:
- Ran 1000 disks for 3000 hours (125 days) => 3 million hours total
- Had 4 failures
- Concluded: 1 failure every 750,000 hours.

But failures aren't memoryless: a disk is more likely to fail at the very beginning of its lifespan, and after it's about five years old. Plus, think about running a large datacenter that has thousands of disks: the probability of a specific disk failing might be low, but the probability of *a* disk failing is high. Even buying a more expensive disk won't fix this issue.

Dealing with sector errors

Suppose a sector of a disk fails. What can we do?
- Silently return the wrong answer
- "Fail fast": detect failure. Each sector has a header with a checksum (e.g., xor all bytes together). Every read fetches a sector and its header. Firmware compares checksum to data. Returns error to driver if checksum doesn't match.
- Correct/mask failure. Re-read if the firmware signals an error

These are general approaches — e.g., could have checksums at the application level (in files), or a backup to recover / correct from a bad file.

Dealing with whole disk failures

So, a lot of things can fail within a disk — sectors, e.g. — but we're going to think about what to do when the *whole* disk fails. This is where replication comes in. We'll start by doing the following:

- On write: write to both disks.
- On read: read from one disk
- Detecting failure: Assume that this is possible; we won't get into it.
- Recovery: after replacing the broken disk, copy data from the good disk over.
What did we gain? We can handle a single-disk failure as long as no two disks fail too close together. We can't handle two disks failing at once, but we're still doing a lot better.

We also would get a performance improvement for reads.

What did we lose? Writes are slower, but not by much; we can issue them in parallel. But we also had to buy a whole new disk, and we didn't get any additional storage capacity.

------------------------------------------------------------------------

RAID
------------------------------------------------------------------------

We can be a lot smarter about this. The technique I described to you is part of a suite of techniques that are part of a system called RAID: Redundant Array of Independent Disks. There are multiple "levels" of RAID; the one I just described was RAID 1.

Here's the basic idea: consider two bits, A and B, and their XOR value, AxB. If I give you any two of those bits, you can recover the third by XOR'ing whatever two blocks together.

If you have A and AxB, Ax(AxB) = B
If you have B and AxB, Bx(AxB) = A
If you have A, B, you're good to go

You can extend this idea to bitstrings; the xor of a bitstring is just the bit-by-bit xor.

Ex: A = 0110; B = 1011; AxB = 1101

We'll extend that idea to disks, which, after all, are just really long bitstrings:
- Take your N disks. Write a sector of data to each of them.
- Add an additional disk (N+1 disks total). Write the XOR of those sector to it. We'll call this disk the parity disk.

Now imagine that one of those disks fails. You can recover its data by just XOR'ing the data from the other disks together.

This technique is RAID 4 (we skipped 2 and 3, because no one uses them in practice. In fact, no one uses RAID 4 in practice; we're coming to RAID 5 in a minute).

So what do we get from RAID 4?

- We can still handle single-disk failures.
- Note that we can recover the parity disk if it fails, too, in
exactly the same way as we'd recover any data disk. Just because it has a special name doesn't make it any more important.

- To store N-disks worth of data, we need N+1 disks, instead of 2N as in RAID 1.

We also see a performance benefit if we stripe a file across multiple disks (you can read in parallel -- we talked about this technique when we talked about improving performance). This has nothing to do with RAID, though; this is just a consequence of having more than one disk on which to store data.

In fact, even though we're focused here on reliability, we never forget performance. RAID 4 does take performance into account in how it does reads and writes. Lower RAID levels write a single file bit-by-bit across a disk, which means that a single read or write has a ton of overhead (since a sector is the smallest unit of reading or writing).

This is the reason we're not discussing RAIDs 2 and 3: they perform poorly and provide no additional realiability. In fact, if you read the original RAID paper, you'll find that it really emphasizes reliability, but that a lot of data organization was driven by performance.

RAID 4 is useful even on single-sector errors, not just full-disk errors: if a single sector fails on a single disk, the disk controller will tell you, and you can recover that sector.

Here's an example of a fault-tolerant system relying on correct code: the disk controller must be correct for all of this to work.

Now, what about downsides? Well, a single write means writing data to two disks, but that seems virtually unavoidable if we want to use replication. The real downside is that every write has to modify the parity disk. Thus, it's hard to issue writes from different applications in parallel, and we're hitting the parity disk more frequently than any other disk.

RAID 5 takes the exact same idea, but spreads the parity sectors out across the N+1 disks. To do this, we need some systematic way of figuring out which disk holds the parity for any given sector, but that's not too hard:

Assume each disk has K sectors, and assume N disks. Let "parity sector i" be the XOR of each disks' sector i. Put parity sector i on disk i mod N.

RAID 5 is used today. So is RAID 6. RAID 6 works on the same principles, but uses a different type of error-correcting code that allows it to recover from two disks failing at once, instead of just
Where we're headed

Even when we think of just replication, you'll find tomorrow that RAID doesn't solve everything.

On a single machine:
- Are we going to try to replicate *everything*? Applications, operating systems, etc., and compare every memory read/write?
  - Performance would be poor, cost would be huge!
- What about failures that aren't independent? RAID can handle a single-disk failure, but imagine a power failure that takes out all N of its disks.

You'll also see tomorrow that RAID is generally not used *across* machines.

But the idea of writing data to multiple places in the system *is* going to stick around. Instead of writing data to multiple places on a single machine, we'll start writing to multiple *machines*.

Immediately, this will lead to some complexity:

- Writes now go over the network. What if it fails?
- If data is frequently updated, how do we keep that consistent? Especially if data is updated by more than one user? You can imagine locks coming back into play, but we had enough trouble getting locking disciplines to work on a single machine!
- If multiple machines start to fail in different ways, how do we know which one has the correct data?

Your Future in 6.033

So here's where we're going.

- Lecture 16: Since replication doesn't solve every type of fault, we'll need a way to deal with these other faults. We will motivate the notion of a "transaction", and talk about what it is, why we use it, and why it helps us deal with fault tolerance.

- Next week: We'll try to get transaction-based systems to perform well and be fault-tolerant on a single machine.
- Week after: We'll get everything to work -- perform well, be fault-tolerant -- across machines.