Notes on Distributed Systems for Young Bloods (/2013/01/14/notes-on-distributed-systems-for-young-bloods/)

January 14, 2013 08:15 AM PST

I've been thinking about the lessons distributed systems engineers learn on the job. A great deal of our instruction is through scars made by mistakes made in production traffic. These scars are useful reminders, sure, but it'd be better to have more engineers with the full count of their fingers.

New systems engineers will find the Fallacies of Distributed Computing (http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing) and the CAP theorem (http://codahale.com/you-cant-sacrifice-partition-tolerance/) as part of their self-education. But these are abstract pieces without the direct, actionable advice the inexperienced engineer needs to start moving[1]. It's surprising how little context new engineers are given when they start out.

Below is a list of some lessons I've learned as a distributed systems engineer that are worth being told to a new engineer. Some are subtle, and some are surprising, but none are controversial. This list is for the new distributed systems engineer to guide their thinking about the field they are taking on. It's not comprehensive, but it's a good beginning.

The worst characteristic of this list is that it focuses on technical problems with little discussion of social problems an engineer may run into. Since distributed systems require more machines and more capital, their engineers tend to work with more teams and larger organizations. The social stuff is usually the hardest part of any software developer's job, and, perhaps, especially so with distributed systems development.

Our background, education, and experience bias us towards a technical solution even when a social solution would be more efficient, and more pleasing. Let's try to correct for that. People are less finicky than computers, even if their interface is a little less standardized.

Alright, here we go.

Distributed systems are different because they fail often. When asked what separates distributed systems from other fields of software engineering, the new engineer often cites latency, believing that's what makes distributed computation hard.

But they're wrong. What sets distributed systems engineering apart is the probability of failure and, worse, the probability of partial failure. If a well-formed mutex unlock fails with an error, we can assume the process is unstable and crash it. But the failure of a distributed mutex's unlock must be built into the lock protocol.

Systems engineers that haven't worked in distributed computation will come up with ideas like "well, it'll just send the write to both machines" or "it'll just keep retrying the write until it succeeds". These engineers haven't completely accepted (though they usually intellectually recognize) that networked systems fail more than systems that exist on only a single machine and that failures tend to be partial instead of total. One of the writes may succeed while the other fails, and so now how do we get a consistent view of the data? These partial failures are much harder to reason about.

Switches go down, garbage collection pauses make masters "disappear", socket writes seem to succeed but have actually failed on the other machine, a slow disk drive on one machines causes a communication protocol in the whole cluster to crawl, and so on. Reading from local memory is simply more stable than reading across a few switches.

Design for failure.

Writing robust distributed systems costs more than writing robust single-machine systems. Creating a robust distributed solution requires more money than a single-machine solution because there are failures that only occur with many machines. Virtual machine and cloud technology make distributed systems engineering cheaper but not as cheap as being able to design, implement, and test on a computer you already own. And there are failure conditions that are difficult to replicate on a single machine. Whether it's because they only occur on dataset sizes much larger than can be fit on a shared machine, or in the network conditions found in datacenters, distributed systems tend to need actual, not simulated, distribution to flush out their bugs. Simulation is, of course, very useful.

Robust, open source distributed systems are much less common than robust, single-machine systems. The cost of running many machines for long periods of time is a burden on open source communities. Hobbyists and dilettantes are the engines of open source software and they do not have the financial resources available to explore or fix many of the problems a distributed system will have. Hobbyists write open source code for fun in their free time and with machines they already own. It's much harder to find open source developers who are willing to spin up, maintain, and pay for a bunch of machines.

Some of this slack has been taken up by engineers working for corporate entities. However, the priorities of their organization may not be in line with the priorities of your organization.

While some in the open source community are aware of this problem, it’s not yet solved. This is hard.

Coordination is very hard. Avoid coordinating machines wherever possible. This is often described as “horizontal scalability”. The real trick of horizontal scalability is independence – being able to get data to machines such that communication and consensus between those machines is kept to a minimum. Every time two machines have to agree on something, the service is harder to implement. Information has an upper limit to the speed it can travel, and networked communication is flakier than you think, and your idea of what constitutes consensus is probably wrong. Learning about the Two Generals (http://en.wikipedia.org/wiki/Two_Generals%27_Problem) and Byzantine Generals (http://en.wikipedia.org/wiki/Byzantine_Generals%27_Problem) problems are useful here. (Oh, and Paxos really is very hard to implement (http://research.google.com/pubs/pub33002.html); that’s not grumpy old engineers thinking they know better than you.)

If you can fit your problem in memory, it’s probably trivial. To a distributed systems engineer, problems that are local to one machine are easy. Figuring out how to process data quickly is harder when the data is a few switches away instead of a few pointer dereferences away. In a distributed system, the well-worn efficiency tricks documented since the beginning of computer science no longer apply. Plenty of literature and implementations are available for algorithms that run on a single machine because the majority of computation has been done on singular, uncoordinated machines. Significantly fewer exist for distributed systems.

“It’s slow” is the hardest problem you’ll ever debug. “It’s slow” might mean one or more of the number of systems involved in performing a user request is slow. It might mean one or more of the parts of a pipeline of transformations across many machines is slow. “It’s slow” is hard, in part, because the problem statement doesn’t provide many clues to location of the flaw. Partial failures, ones that don’t show up on the graphs you usually look up, are lurking in a dark corner. And, until the degradation becomes very obvious, you won’t receive as many resources (time, money, and tooling) to solve it. Dapper (http://research.google.com/pubs/pub36356.html) and Zipkin (http://engineering.twitter.com/2012/06/distributed-systems-tracing-with-zipkin.html) were built for a reason.

Implement backpressure throughout your system. Backpressure is the signaling of failure from a serving system to the requesting system and how the requesting system handles those failures to prevent overloading itself and the serving system. Designing for backpressure means bounding resource utilization during times of overload and times of system failure. This is one of the basic building blocks of creating a robust distributed system.

Common versions include dropping new messages on the floor (and incrementing a metric) if the system’s resources are already over-scheduled, and shipping errors back to users when the system determines it will be unable to finish the request in a given amount of time. Timeouts and exponential back-offs on connections and requests to other systems are also useful.

Without backpressure mechanisms in place, cascading failure or unintentional message loss become likely. When a system is not able to handle the failures of another, it tends to emit failures to another system that depends on it.

Find ways to be partially available. Partial availability is being able to return some results even when parts of your system is failing.

Search is an ideal case to explore here. Search systems trade-off between how good their results are and how long they will keep a user waiting. A typical search system sets a time limit on how long it will search its documents, and, if that time limit expires before all of its documents are searched, it will return whatever results it has gathered. This makes search easier to scale in the face of intermittent slowdowns, and errors because those failures are treated the same as not being able to search all of their documents. The system allows for partial results to be returned to the user and its resilience is increased.

And consider a private messaging feature in a web application. We easily believe that if private messaging is down, the image upload feature should probably keep working. So, consider designing for partial failure in the private messaging service itself. This takes some thought, of course. People are generally more okay with private messaging being down for them (and maybe some other users) than they are with all users having some of their messages go missing. If the service is overloaded or one of its machines are down, failing out just a small fraction of the userbase is preferable to missing data for a larger fraction. Being able to recognize these kinds of trade-offs in partial availability is good to have in your toolbox.

Metrics are the only way to get your job done. Exposing metrics (such as latency percentiles, increasing counters on certain actions, rates of change) is the only way to cross the gap from what you believe your system does in production and what it actually is doing. Knowing how the system's behavior on day 20 is different from its behavior on day 15 is the difference between successful engineering and failed shamanism. Of course, metrics are necessary to understand problems and behavior, but they are not sufficient to know what to do next.

A diversion into logging. Log files are good to have, but they tend to lie. For example, it's very common for the logging of a few error classes to take up a large proportion of a space in a log file but, in actuality, occur in a very low proportion of requests. Because logging successes is redundant in most cases (and would blow out the disk in most cases) and because engineers often guess wrong on which kinds of error classes are useful to see, log files get filled up with all sorts of odd bits and bobs. Prefer logging as if someone who has not seen the code will be reading the logs.

I've seen a good number of outages extended by another engineer (or myself) over-emphasizing something odd we saw in the log without first checking it against the metrics. I've also seen another engineer (or myself) Sherlock-Holmes'ing an entire set of failed behaviors from a handful of log lines. But note: a) we remember those successes because they are so very rare and b) you're not Sherlock unless the metrics or the experiments back up the story.

Use percentiles, not averages. Percentiles (50th, 99th, 99.9th, 99.99th) are more accurate and informative than averages in the vast majority of distributed systems. Using a mean assumes that the metric under evaluation follows a bell curve but, in practice, this describes very few metrics an engineer cares about. "Average latency" is a commonly reported metric, but I've never once seen a distributed system whose latency followed a bell curve. If the metric doesn't follow a bell curve, the average is meaningless and leads to incorrect decisions and understanding. Avoid the trap by talking in percentiles. Default to percentiles, and you'll better understand how users really see your system.

Learn to estimate your capacity. You'll learn how many seconds are in a day because of this. Knowing how many machines you need to perform a task is the difference between a long-lasting system, and one that needs to be replaced 3 months into its job. Or, worse, needs to be replaced before you finish productionizing it.

Consider tweets. How many tweet ids can you fit in memory on a common machine? Well, a typical machine at the end of 2012 has 24 GB of memory, you'll need an overhead of 4-5 GB for the OS, another couple, at least, to handle requests, and a tweet id is 8 bytes. This is the kind of back of the envelope calculation you'll find yourself doing. Jeff Dean's Numbers Everyone Should Know (http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf) slide is a good expectation-setter.

Feature flags are how infrastructure is rolled out. "Feature flags" are a common way product engineers roll out new features in a system. Feature flags are typically associated with frontend A/B testing where they are used to show a new design or feature to only some of the userbase. But they are a powerful way of replacing infrastructure as well.

Suppose you're going from a single database to a service that hides the details of a new storage solution. Have the service wrap around the legacy storage, and ramp up writes to it slowly. With backfilling, comparison checks on read (another feature flag), and then slow ramp up of reads (yet another flag), you will have much more confidence and fewer disasters. Too many projects have failed because they went for the "big cutover" or a series of "big cutovers" that were then forced into rollbacks by bugs found too late.

Feature flags sound like a terrible mess of conditionals to a classically trained object-oriented developer or a new engineer with well-intentioned training. And the use of feature flags means accepting that having multiple versions of infrastructure and data is a norm, not an rarity. This is a deep lesson. What works well for single-machine systems sometimes falters in the face of distributed problems.

Feature flags are best understood as a trade-off, trading local complexity (in the code, in one system) for global simplicity and resilience.

Choose id spaces wisely. The space of ids you choose for your system will shape your system.

The more ids required to get to a piece of data, the more options you have in partitioning the data. The fewer ids required to get a piece of data, the easier it is to consume your system's output.

Consider version 1 of the Twitter API. All operations to get, create, and delete tweets were done with respect to a single numeric id for each tweet. The tweet id is a simple 64-bit number that is not connected to any other piece of data. As the number of tweets goes up, it becomes clear that creating user tweet timelines and the timeline of other user's subscriptions may be efficiently constructed if all of the tweets by the same user were stored on the same machine.

But the public API requires every tweet be addressable by just the tweet id. To partition tweets by user, a lookup service would have to be constructed. One that knows what user owns which tweet id. Doable, if necessary, but with a non-trivial cost.

An alternative API could have required the user id in any tweet look up and, initially, simply used the tweet id for storage until user-partitioned storage came online. Another alternative would have included the user id in the tweet id itself at the cost of tweet ids no longer being k-sortable and numeric.

Watch out for what kind of information you encode in your ids, explicitly and implicitly. Clients may use the structure of your ids to de-anonymize private data, crawl your system in ways you didn't expect (auto-incrementing ids are a typical sore point), or a host of other things (https://www.owasp.org/index.php/Top_10_2010-A4-Insecure_Direct_Object_References) you won't expect.

Exploit data-locality. The closer the processing and caching of your data is kept to its persistent storage, the more efficient your processing, and the easier it will be to keep your caching consistent and fast. Networks have more failures and more latency than pointer dereferences and `fread(3)`.

Of course, data-locality implies locality in space, but also locality in time. If multiple users are making the same expensive request at nearly the same time, perhaps their requests can be joined into one. If multiple instances of requests for the same kind of data are made near to one another, they could be joined into one larger request. Doing so often affords lower communication overheard and easier fault management.

Writing cached data back to persistent storage is bad. This happens in more systems than you'd think. Especially ones originally designed by people less experienced in distributed systems. Many systems you'll inherit will have this flaw. If the implementers talk about "Russian-doll caching", you have a large chance of hitting highly visible bugs. This entry could have been left out of the list, but I have a special hate in my heart for it. A common presentation of this flaw is user information (e.g. screennames, emails, and hashed passwords) mysteriously reverting to a previous value.

Computers can do more than you think they can. In the field today, there's plenty of misinformation about what a machine is capable of from practitioners that do not have a great deal of experience.

At the end of 2012, a light web server had 6 or more processors, 24 GB of memory and more disk space than you can use. A relatively complex CRUD (http://en.wikipedia.org/wiki/Create,_read,_update_and_delete) application in a modern language runtime on a single machine is trivially capable of doing thousands of requests per second within a few hundred milliseconds. And that's a deep lower bound. In terms of operational ability, hundreds of requests per second per machine is not something to brag about in most cases.

Greater performance is not hard to come by, especially if you are willing to profile your application and introduce efficiencies based on your measurements.

Use the CAP theorem to critique systems. The CAP theorem isn't something you can build a system out of. It's not a theorem you can take as a first principle and derive a working system from. It's much too general in its purview, and the space of possible solutions too broad.

However, it is well-suited for critiquing a distributed system design, and understanding what trade-offs need to be made. Taking a system design and iterating through the constraints CAP puts on its subsystems will leave you with a better design at the end. For homework, apply the CAP theorem's constraints to a real world implementation of Russian-doll caching.

One last note: Out of C, A, and P, you can't choose CA (http://codahale.com/you-cant-sacrifice-partition-tolerance/).

Extract services. "Service" here means "a distributed system that incorporates higher-level logic than a storage system and typically has a request-response style API". Be on the lookout for code changes that would be easier to do if the code existed in a separate service instead of in your system.

An extracted service provides the benefits of encapsulation typically associated with creating libraries. However, extracting out a service improves on creating libraries by allowing for changes to be deployed faster and easier than upgrading the libraries in its client systems. (Of course, if the extracted service is hard to deploy, the client systems are the ones that become easier to deploy.) This ease is owed to the fewer code and operational dependencies in the smaller, extracted service and the strict boundary it creates makes it harder to "take shortcuts" that a library allows for. These shortcuts almost always make it harder to migrate the internals or the client systems to new versions.

The coordination costs of using a service is also much lower than a shared library when there are multiple client systems. Upgrading a library, even with no API changes needed, requires coordinating deploys of each client system. This gets harder when data corruption is possible if the deploys are performed out of order (and it's harder to predict that it will happen). Upgrading a library also has a higher social coordination cost than deploying a service if the client systems have different maintainers. Getting others aware of and willing to upgrade is surprisingly difficult because their priorities may not align with yours.

The canonical service use case is to hide a storage layer that will be undergoing changes. The extracted service has an API that is more convenient, and reduced in surface area compared to the storage layer it fronts. By extracting a service, the client systems don't have to know about the complexities of the slow migration to a new storage system or format and only the new service has to be evaluated for bugs that will certainly be found with the new storage layout.

There are a great deal of operational and social issues to consider when doing this. I cannot do them justice here. Another article will have to be written.

[1] Of course, Rotem-Gal-Oz's take on the fallacies (http://www.rgoarchitects.com/Files/fallacies.pdf) is very good.

Much love to my reviewers Bill de hÓra (https://twitter.com/dehora), Coda Hale (https://twitter.com/coda), JD Maturen (https://twitter.com/jdmaturen), Micaela McDonald (https://twitter.com/nora), and Ted Nyman (https://twitter.com/tnm). Your insight and care was invaluable.