Getting Real About Distributed System Reliability

There is a lot of hype around distributed data systems, some of it justified. It’s true that the internet has centralized a lot of computation onto services like Google, Facebook, Twitter, LinkedIn (my own employer), and other large web sites. It’s true that this centralization puts a lot of pressure on system scalability for these companies. Its true that incremental and horizontal scalability is a *deep* feature that requires redesign from the ground up and can’t be added incrementally to existing products. It’s true that, if properly designed, these systems can be run with no *planned* downtime or maintenance intervals in a way that traditional storage systems make harder. It’s also true that software that is explicitly designed to deal with machine failures is a very different thing from traditional infrastructure. All of these properties are critical to large web companies, and are what drove the adoption of horizontally scalable systems like Hadoop, Cassandra, Voldemort, etc. I was the original author of Voldemort and have worked on distributed infrastructure for the last four years or so. So in-so-far as there is a “big data” debate, I am firmly in the “pro-” camp. But one thing you often hear is that this kind of software is more reliable than the traditional alternatives it replaces, and this just isn’t true. It is time people talked honestly about this.

You hear this assumption of reliability everywhere. Now that scalable data infrastructure has a marketing presence, it has really gotten bad. Hadoop or Cassandra or what-have-you can tolerate machine failures then they must be unbreakable right? Wrong.

Where does this come from? Distributed systems need to partition data or state up over lots of machines to scale. Adding machines increases the probability that some machine will fail, and to address this these systems typically have some kind of replicas or other redundancy to tolerate failures. The core argument that gets used for these systems is that if a single machine has probability P of failure, and if the software can replicate data N times to survive N-1 failures, and if the machines fail *independently*, then the probability of losing a particular piece of data must be P^N. So for any desired reliability R and any single-node failure probability P you can pick some replication N so that $P^N < R$. This argument is the core motivation behind most variations on replication and fault tolerance in distributed systems. It is true that without this property the system would be hopelessly unreliable as it grew. But this leads people to believe that distributed software is somehow innately reliable, which unfortunately is utter hogwash.

Where is the flaw in the reasoning? Is it the dreaded Hadoop single points of failure? No, it is far more fundamental than that: the problem is the assumption that failures are independent. Surely no belief could possibly be more counter to our own experience or just common sense than believing that there is no correlation between failures of machines in a cluster. You take a bunch of pieces of identical hardware, run them on the same network gear and power systems, have the same people run and manage and configure them, and run the same (buggy) software on all of them. It would be incredibly unlikely that the failures on these machines would be independent of one another in the probabilistic sense that motivates a lot of distributed infrastructure. If you see a bug on one machine, the same bug is on all the machines. When you push bad config, it is usually game over no matter how many machines you push it to.

P^N is an upper bound on reliability but one that you could never, never approach in practice. For example Google has a fantastic paper that gives empirical numbers on system failures in Bigtable and GFS and reports empirical data on groups of failures that show rates several orders of magnitude higher than the independence assumption would predict. This is what one of the best system and operations teams in the world can get: your numbers may be far worse.

The actual reliability of your system depends largely on how bug free it is, how good you are at monitoring it, and how well you have protected against the myriad issues and problems it has. This isn't any different from traditional systems, except that the new software is far less mature. I don't mean this disparagingly, I work in this area, it is just a fact. Maturity comes with time and usage and effort. This software hasn't been around for as long as MySQL or Oracle, and worse, the expertise to run it reliably is much less common. MySQL and Oracle administrators are plentiful, but folks experience with, say, serious production Zookeeper operations knowledge are much more rare.

Kernel filesystem engineers say it takes about a decade for a new filesystem to go from concept to maturity. I am not sure these systems will be mature much faster—they are not easier systems to build and the fundamental design space is much less well explored. This doesn't mean they won't be *useful*sooner, especially in domains where they solve a pressing need and are approached with an appropriate amount of caution, but they are not yet by any means a mature technology.

Part of the difficulty is that distributed system software is actually quite complicated in comparison to single-server code. Code that deals with failure cases and is "cluster aware" is extremely tricky to get right. The root of the problem is that dealing with failures effectively explodes the possible state space that needs testing and validation. For example it doesn't even make sense to expect a single-node database to be fast if its disk system suddenly gets really slow (how could it), but a distributed system does need to carry on in the presence of single degraded machine because it has some many machines, one is sure to be degraded. These kind of "semi-failures" are common and very hard to deal with. Correctly testing these kinds of issues in a realistic setting is brutally hard and the newer generation of software doesn't have anything like the QA processes its more mature predecessors had. (If you get a chance get someone who has worked at Oracle to describe to you what kind of testing they do to a line of code that goes into their database before it gets shipped to customers). As a result there are a lot of bugs. And of course these bugs are on all the machines, so they absolutely happen together.

Likewise distributed systems typically require more configuration and more complex configuration because they need to be cluster aware, deal with timeouts, etc. This configuration is, of course, shared; and this creates yet another opportunity to bring everything to its knees.

And finally these systems usually want lots of machines. And no matter how good you are, some part of operational difficulty always scales with the number of machines.

Let's discuss some real issues. We had a bug in Kafka recently that lead to the server incorrectly interpreting a corrupt request as a corrupt log, and shutting itself down to avoid appending to a corrupt log. Single machine log corruption is the kind of thing that should happen due to a disk error, and bringing down the corrupt node is the right behavior—it shouldn't happen on all the machines at the same time unless all the disks fail at once. But since this was due to corrupt *requests*, and since we had one client that sent corrupt requests, it was able to sequentially bring down all the servers. Oops. Another example is this Linux bug which causes the system to crash after ~200 days of uptime. Since machines are commonly restarted sequentially this lead to a situation where a large percentage of machines went hard down one after another. Likewise any memory management problems—either leaks or GC problems—tend to happen everywhere at once or not at all. Some companies do public post-mortums for major failures and these are a real wealth of failures in systems that aren't supposed to fail. This paper has an excellent summary of HDFS availability at Yahoo—they note how few of the problems are of the kind that high availability for the namenode would solve. This list could go on, but you get the idea.

I have come around to the view that the real core difficulty of these systems is operations, not architecture or design. Both are important but good operations can often work around the limitations of bad (or incomplete) software, but good software cannot run reliably with bad operations. This is quite different from the view of unbreakable, self-healing, self-operating systems that I see being pitched by the more enthusiastic NoSQL hypesters. Worse yet, you can't easily buy good operations in the same way you can buy good software—you might be able to hire good people (if you can find them) but this is more than just people; it is practices, monitoring systems, configuration management, etc.

These difficulties are one of the core barriers to adoption for distributed data infrastructure. LinkedIn and other companies that have a deep interest in doing creative things with data have taken on the burden of building this kind of expertise in-house—we employ committers on Hadoop and other open source projects on both our engineering and operations team, and have done a lot of from-scratch developmentin this space where there was gaps. This makes it feasible to take full advantage of an admittedly valuable but immature set of technologies, and let's us build products we couldn't otherwise—but this kind of investment only makes sense at a certain size and scale. It may be too high a cost for small startups or companies outside the web space trying to bootstrap this kind of knowledge inside a more traditional IT organization.

This is why people should be excited about things like Amazon's DynamoDB. When DynamoDB was released, the company DataStax that supports and leads development on Cassandra released a feature comparison checklist. The checklist was unfair in many ways (as these kinds of vendor comparisons usually are), but the biggest thing missing in the comparison is that *you* don't run DynamoDB, Amazon does. That is a huge, huge difference. Amazon is good at this stuff, and has shown that they can (usually) support massively multi-tenant operations with reasonable SLAs, *in practice*.

I really think there is really only one thing to talk about with respect to reliability: continuous hours of successful production operations. That's it. In some ways the most obvious thing, but not typically what you hear when people talk about these systems. I will believe the system can tolerate (some) failures when I see it tolerate those failures; I believe it can run for a year without downtime when i see it run for a year without downtime. I call this empirical reliability (as opposed to theoretical reliability). And getting good empirical reliability is really, really hard. These systems end up being large hunks of monitoring, tests, and operational procedures with a teeny little distributed system strapped to the back.

You see this showing up in discussions of the CAP theorem all the time. The CAP theorem is a useful thing, but it applies more to system design than system implementations. A design is simple enough that you can maybe prove it provides consistency or tolerates partition failures under some assumptions. This is a useful lens to look at system designs. You can't hope to do this kind of proof with an actual system implementation, the thing you run. The difficulty of building these things means it is really unthinkable that these systems are, in actual reality, either consistent, available, *or* partition tolerant—they certainly all have numerous bugs that will break each of these guarantees. I really like this paper that takes the approach of actually trying to calculate the observed consistency of eventually consistent systems—they seem to do it via simulation rather than measurement, which is unfortunate, but the idea is great.

It isn't that system design is meaningless, it is worth discussing the system design as it does act as a kind of limiting factor on certain aspects of reliability and performance as the implementation matures and improves, but don't take it too seriously as guaranteeing *anything*.

So why isn't this kind of empirical measurement more talked about? I don't know. My pet theory is it has to do with the somewhat rigid and deductive mindset of classical computer science. This is inherited from pure math, and conflicts with the attitude in scientific disciplines. It leads to a preference for starting with axioms, and then proving various properties that follow from these axioms. This world view doesn’t embrace the kind of empirical measurements you would expect to justify claims about reality (for another example see this great blog post on programming language productivity claims in programming language research). But that is getting off topic. Suffice it to say, when making predictions about how a system will work in the real world I believe in measurements of reality a lot more than arguments from first-principles.

I think we should insist on a little more rigor and empiricism in this area.

I would love to see claims in academic publication around practicality or reliability justified in the same way we justify performance claims–by doing it. I would be a lot more likely to believe an academic distributed system was practically feasible if it was run continuously under load for a year successfully and if information was reported on failures and outages. Maybe that isn’t feasible for an academic project, but few other allegedly scientific academic disciplines can get away with making claims about reality without evidence.

More broadly I would like to see more systems whose *operation* is verifiable. Many systems have the ability to log out information about their state in a way that makes programmatic checking of various invariants and guarantees possible (such as consistency or availability). An example of such an invariant for a messaging system is that all the messages sent to it are received by all the subscribers. We actually measure the truth of this statement in real-time in production for our Kafka data pipeline for all 450 topics and all their subscribers. The number of corner-cases one uncovers with this kind of check, run through a few hundred use cases and a few billion messages per day is truly astounding. I think this is a special case of a broad class of verification that can be done on the running system that goes far far deeper than what is traditionally considered either monitoring or testing. Call it unit testing in production, or self-proving systems, or next generation monitoring, or whatever, but I think this kind of deep verification is something that makes turning the theoretical claims a design makes into measured properties of the real running system.

Likewise if you have a “NoSQL vendor” I think it is reasonable to ask them to provide hard information on customer outages. They don’t need to tell you who the customer is, but they should let you know the observed real-life distribution of MTTF and MTTR they are able to achieve, not just highlight one or two happy cases. Make sure you understand how they measure this, do they have automated test load that runs or just wait for people to complain? This is a reasonable thing for people paying for a service to ask for. To a certain extent if they provide this kind of empirical data it isn’t clear why you should even *care*what their architecture is beyond intellectual curiosity.

Distributed systems solve a number of key problems at the heart of scaling large websites. I absolutely think this is going to become the default approach to handling state in internet application development. But no one benefits from the kind of irrational exuberance that currently surrounds the “big data” or nosql systems communities. This area is experiencing a boom—but nothing takes us from boom to bust like unrealistic hype and broken promises. We should be a little more honest about where these systems already shine and where they are still maturing.