acmqueue Too Big to Fail

Visibility leads to debuggability

Dear KV,

Our project has been rolling out a well-known, distributed key/value store onto our infrastructure, and we've been surprised—more than once—when a simple increase in the number of clients has not only slowed things, but brought them to a complete halt. This then results in rollback while several of us scour the online forums to figure out if anyone else has seen the same problem. The entire reason for using this project's software is to increase the scale of a large system, so I have been surprised at how many times a small increase in load has led to a complete failure. Is there something about scaling systems that's so difficult that these systems become fragile, even at a modest scale?

Scaled Back

Dear Scaled,

If someone tells you that scaling out a distributed system is easy they are either lying or drunk, and possibly both. Anyone who has worked with distributed systems for more than a week should have this knowledge integrated into how they think, and if not, they really should start digging ditches. Not to say that ditch digging is easier but it does give you a nice, focused task that's achievable in a linear way, based on the amount of work that you put into it. Distributed systems, on the other hand, react to increases in offered load in what can only politely be referred to as non-deterministic ways. If you think programming a single system is hard, programming a distributed system is a nightmare of Orwellian proportions where you almost are forced to eat rats if you want to join the party.

Non-distributed systems fail in much more predictable ways. Tax a single system and you run out of memory, or CPU, or disk space, or some other resource, and the system has little more than a snow ball's chance of surviving a Hawaiian holiday. The parts of the problem are so much closer together and the communication between those components is so much more reliable that figuring out "who did what to whom" is tractable. Unpredictable things can happen when you overload a single computer, but you generally have complete control over all of the resources involved. Run out of RAM? Buy more. Run out of CPU, profile and fix your code. Too much data on disk? Buy a bigger one. Moore's law is still on your side in many cases, giving you double the resources every 18 months.

The problem is that eventually you will probably want a set of computers to implement your target system. Once you go from one computer to two, it's like going from a single child to two children. To paraphrase an old comedy sketch, if you only have one child, it's not the same as having two or more children. Why? Because when you have one child and all the cookies are gone from the cookie jar, YOU KNOW WHO DID IT! Once you have two or more children, each has some level of plausible deniability. They can, and will, lie to get away with having eaten the cookies. Short of slipping your kids a truth serum at breakfast every morning, you have no idea who is

telling the truth and who is lying. The problem of truthfulness in communication has been heavily studied in computer science, and yet we still do not have completely reliable ways to build large distributed systems.

One way that builders of distributed systems have tried to address this problem is to put in somewhat arbitrary limits to prevent the system from ever getting too large and unwieldy. The distributed key store Redis had a limit of 10,000 clients that could connect to the system. Why 10,000? No clue, it's not even a typical power of 2. One might have expected 8,192 or 16,384, but that's probably another article. Perhaps the authors had been reading the *Tao Te Ching* and felt that their universe only needed to contain 10,000 things. Whatever the reason, this seemed like a good idea at the time.

Of course, limiting the number of clients is only one way of protecting a distributed system against overload. What happens when a distributed system moves from running on 1Gbps network hardware to 10Gbps NICs? Moving from 1Gbps to 10Gbps doesn't "just" increase the bandwidth by an order of magnitude, it also reduces the request latency. Can a system with 10,000 nodes move smoothly from 1G to 10G? Good question, you'd need to test or model that, but it's pretty likely that a single limitation—such as number of clients—is going to be insufficient to prevent the system from getting into some very odd situations. Depending on how the overall system decides to parcel out work, you might wind up with hot spots, places where a bunch of requests all get directed to a single resource, effectively creating what looks like a denial of service attack and destroying a node's effective throughput. The system will then fail out that node and redistribute the work again, perhaps picking another target, and taking it out of the system because it looks like it, too, has failed. In the worst case, this continues until the entire system is brought to its knees and fails to make any progress on solving the original problem that was set for it.

Distributed systems that use a hash function to parcel out work are often dogged by this problem. One way to judge a hash function is by how well-distributed the results of the hashing function are, based on the input. A good hash function for distributing work would parcel out work completely evenly to all nodes based on the input, but having a good hash function isn't always good enough. You might have a great hash function, but feed it poor data. If the source data fed into the hash function doesn't have sufficient diversity (that is, it is relatively static over some measure, such as requests) then it doesn't matter how good the function is, as it still won't distribute work evenly over the nodes.

Take, for example, the traditional networking 4-tuple, source and destination IP address, and source and destination port. Together this is 96 bits of data, which seems like a reasonable amount of data to feed the hashing function. In a typical networking cluster, the network will be one of the three well-known RFC 1918 addresses (192.168.0.0/16, 172.16.0.0/12, or 10.0.0.0/8). Let's imagine a network of 8,192 hosts, because I happen to like powers of 2. Ignoring subnetting completely, we assign all 8,192 hosts addresses from the 192.168.0.0 space, numbering them consecutively 192.168.0.1-192.168.32.1. The service being requested has a constant destination port number (e.g., 6379) and the source port is ephemeral. The data we now put into our hash function are the two IPs and the ports. The source port is pseudo-randomly chosen by the system at connection time from a range of nearly 16 bits. It's nearly 16 bits because some parts of the port range are reserved for privileged programs, and we're building an underprivileged system. The destination port is constant, so we remove 16 bits of change from the input to the function. Those nice fat IPv4 addresses that

should be giving us 64 bits of data to hash on actually only give us 13 bits, because that's all we need to encode 8,192 hosts. The input to our hashing function isn't 96 bits, but is actually fewer than 42. Knowing that, you might pick a different hash function or change the inputs, inputs that really do lead to the output being spaced evenly over our hosts. How work is spread over the set of hosts in a distributed system is one of the main keys to whether that system can scale predictably, or at all.

An exhaustive discussion of how to scale distributed systems is a topic for a book far longer than this piece, but we can't leave the topic until we talk about what debugging features exist in the distributed system. "The system is slow" is a poor bug report—in fact, it is useless. However, it is the one most often uttered in relation to distributed systems. Typically the first thing that users of the system notice is that the response time has increased and that the results they get from the system take far longer than normal. A distributed system needs to express, in some way, its local and remote service times so that the systems operators, such as the devops or systems administration teams, can track down the problem. Hot spots can be found through the periodic logging of the service request arrival and completion on each host. Such logging needs to be lightweight and not directed to a single host, which is a common mistake. When your system gets busy and the logging output starts taking out the servers, that's bad. Recording system level metrics, including CPU, memory and network utilization will also help in tracking down problems, as will the recording of network errors. If the underlying communications medium becomes overloaded, this may not show up on a single host, but will result in a distributed set of errors, with a small number at each node, which lead to chaotic effects over the whole system. Visibility leads to debuggability; you cannot have the latter without the former.

Coming back around to your original point, I am not surprised that small increases in offered load are causing your distributed system to fail, and, in fact, I am most surprised that some distributed systems work at all. Making the load, hot spots, and errors visible over the system may help you track down the problem and continue to scale it out even further. Or, you may find that there are limits to the design of the system you are using, and you'll have to either choose another or write your own. I think you can see now why you might want to avoid the latter at all costs.

KV

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

KODE VICIOUS, known to mere mortals as George V. Neville-Neil, works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, and rewriting your bad code (OK, maybe not that last one). He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. He is an avid bicyclist and traveler who currently lives in New York City.

© 2014 ACM 1542-7730/14/1100 \$10.00