Managing Systems in an Age of Dynamic Complexity

Or: Why Does My Single 2U Server Have Better Uptime than GCP?

LAURA NOLAN

Laura Nolan's background is in site reliability engineering, software engineering, distributed systems, and computer science. She wrote the "Managing Critical State" chapter in the O'Reilly Site Reliability Engineering book and was co-chair of SREcon18 Europe/Middle East/Africa. Laura Nolan is a production engineer at Slack. laura.nolan@gmail.com

A decade ago most systems administrators were running relatively static systems. We had servers which were provisioned and updated by hand, or maybe via a human invoking Puppet, Capistrano, or CFEngine. Instances were typically added to load balancer pools manually. New instances would be provisioned when administrators decided that more capacity was needed, and systems were sized for peak loads (plus a margin). Networks were configured by hand.

This kind of static administration has pros and cons. Servers that stay around a long time and get updated manually can be very hard to replace when they fail. Instances that should be identically configured can experience drift, leading to hard-to-diagnose production problems. There's a lot of work to be done by hand. Load balancing and network failover can give some capacity to handle failure, but often you'll need to alert a human to fix problems. Even if the system is robust enough to stay up in the face of failure, someone usually needs to fix things afterwards to bring things back to the intended capacity.

Fundamentally, a human or a team of humans is operating the system directly. When something changes in the system, it is because someone intended it to change or because something failed. Small, static, human-managed systems can have really good uptime. Change tends to be relatively infrequent and human initiated, so it can usually be undone quickly if problems arise. Hardware failures are rare because the likelihood of failure is proportional to the amount of hardware you have: individual server uptimes measured in years aren't too uncommon, although nowadays long-lived servers are generally considered an antipattern as Infrastructure-as-Code has become popular.

At scale, things change. All the downsides of managing a lot of servers by hand become much worse: the human toil, the pager noise. High uptime becomes hard to maintain as failures become more common. Cost becomes a factor: there is almost certainly going to be organizational pressure to be as efficient as possible in terms of computing resources. System architectures are likely to be complex microservice meshes, which are much more challenging to manage than simpler monoliths.

These pressures lead to the rise of what I will term dynamic control systems: those systems where the jobs once done by human administrators, such as provisioning instances, replacing failed instances, restarting jobs, applying configuration, and updating load balancer back-end pools, are done by machines. The most obvious examples of today's dynamic systems are software-defined networking (SDN), job or container orchestration, and service meshes. These systems do indeed work well to increase resource utilization and thus reduce costs, to prevent human toil scaling with the size of the services managed, and to allow systems to recover from routine hardware failures without human intervention.

Dynamic control systems, however, also bring completely novel challenges to system operators. Most dynamic control systems have a similar structure, shown in Figure 1.

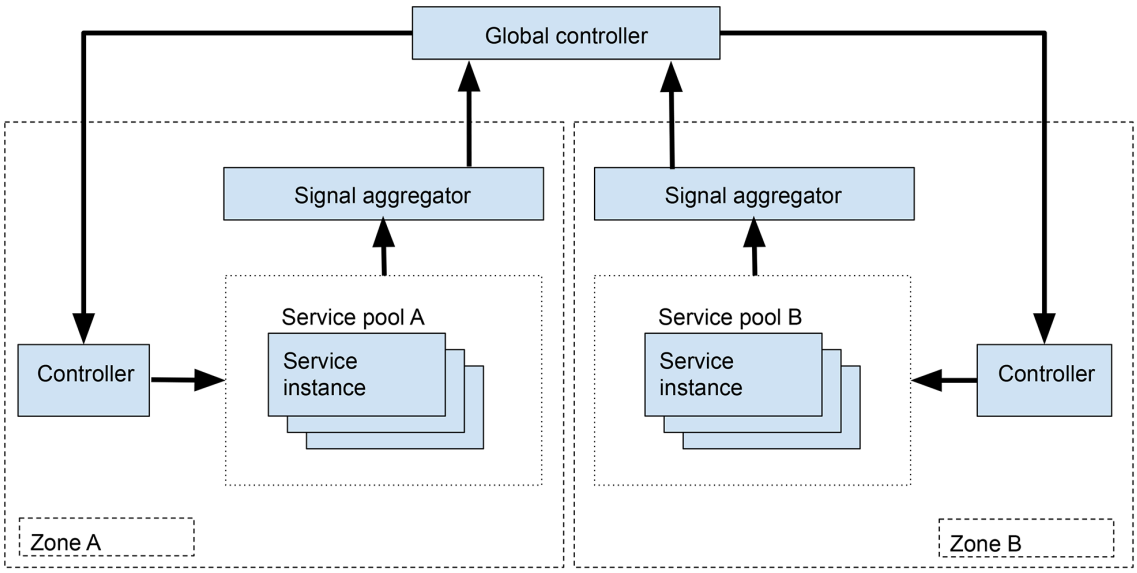

Figure 1: A generic dynamic control system architecture, showing two separate sets of service instances

The components are:

- Service pools: a set of instances doing work of some kind.
- Signal aggregator: a service which collects metrics from the service pool instances (often a monitoring system such as Prometheus), usually one per "zone" (meaning region, datacenter, cluster, availability zone—whichever domain makes sense for a given service).
- Global controller: a service which receives signals from the signal aggregator and makes global decisions about configuration of the service.
- Controller: a service which receives updates from the global controller and applies configuration locally.

Google's B4 SDN WAN control plane [1] is an example of this architecture. Variants exist, of course: a small dynamic control system might merge some of these functions into fewer services—for instance, a service running in just one zone might collapse the functions of the aggregator, controller, and global controller into a single service. Another possible architecture is to have independent controllers in each zone for services that don't need global coordination.

There might be multiple controllers (for example, mapping this concept to Kubernetes, the global controller is the Kubernetes Master and the controllers are the kubelets). Most SDN architectures are a variant on this, as are coordinated load-balancing and rate-limiting architectures, including service meshes. Many job orchestration functions like autoscaling and progressive rollouts/canarying are structured in this way, too.

The dynamic control system architecture above is popular because it works: it scales, allowing globally optimal decisions or configurations to be computed and pushed back to instances quite quickly. The controllers provide useful telemetry as well as a point of control to apply overrides or other exceptions.

However, it also has its downsides. Most notably, it adds a lot more software components, all of which can themselves fail or misbehave. A naive implementation of a global load-balancing control plane which experiences correlated failure in its zonal monitoring subsystems could easily lead to global failure, if its behavior in such cases is not carefully thought through or if it has bugs.

Another critical weakness of dynamic systems architecture is that it distances operators from the state of their systems: we do not make changes directly anymore. We understand normal operation less well, and it can also be harder to understand and fix abnormal operation. Charles Perrow discusses this phenomenon in *Normal Accidents* [2] in the context of the Apollo 13 accident. Mission control had detailed telemetry but was confused about the nature of the incident. The astronauts knew that they had just initiated an operation on a gas tank, they felt a jolt and they saw liquid oxygen venting. Their proximity to the system was key to their understanding.

Systems administrators used to be more like the astronauts, but now our profession is moving towards being mission control. We are now in the business of operating the systems that operate the systems, which is a significantly harder task. In addition to managing, monitoring, and planning for failure in our core systems, we must now also manage, monitor, and plan for the failure of our dynamic control planes. Worse again, we normally have multiple dynamic control planes doing different tasks. Figuring

out all the potential interactions between multiple control planes in working order is probably impossible; trying to figure out how multiple control planes might interact when one or more of them have bugs or experience failure is definitely impossible.

This brings us back to the subtitle of this article: how can a single server have better uptime than a cloud platform which is carefully designed by competent engineers for availability in the case of failure? Let's examine two outages.

On April 11, 2016, Google Compute Engine (GCE) lost external connectivity for 18 minutes. The RCA (root cause analysis) for the incident [3] is a study in dynamic control systems failure:

1. An unused IP block was removed from a network configuration, and the control system that propagates network configurations began to process it. A race condition triggered a bug which removed all GCE IP blocks.
2. The configuration was sent to a canary system (a second dynamic control system), which correctly identified a problem, but the signal it sent back to the network configuration propagation system wasn't correctly processed.
3. The network configuration was rolled out to other sites in turn. GCE IP blocks were advertised (over BGP) from multiple sites via IP Anycast. One could take the view that BGP advertisements themselves constitute a third dynamic control system. This means that probes to these IPs continued to work until the last site was withdrawn—see [4] for more detail on why. This meant the rollout process lacked critical signal on the effect of its actions on the health of GCE.

This incident features multiple control systems with multiple failures in processing and in monitoring. These systems are utterly necessary to manage networks at this scale, but it is also impossible to predict the many ways in which they can go wrong. The following is a classic complex systems failure [5]:

On June 2, 2019, Google Cloud experienced serious network degradation for over three hours. The RCA [6] is another tale of dynamic control systems misadventure in which many instances of the network control plane system were accidentally descheduled by the control system responsible for managing datacenter maintenance events. It took two misconfigurations and a software bug for that to happen: again, there is no way to predict that specific sequence of events. This incident is also an example of the difficulty that can arise in restoring control system state when it has been lost or corrupted.

Dynamic control systems are inherently complex, and will always be challenging, but it is to be hoped that best practices regarding their operation will emerge. One such best practice that is often suggested is to avoid systems that can make global changes, but that is not always easy. Some systems are inherently global, anycast networks being a good example as well as systems that balance load across multiple datacenters or regions.

This is one of the key challenges of modern large systems administration, SRE, and DevOps: human-managed static systems don't scale, and we haven't yet developed enough experience with dynamic control systems to run them as reliably as our 2U server of yore—and maybe we'll never be able to make them as reliable.

Both of the incidents analyzed here are Google RCAs, but dynamic control system problems are by no means unique to Google (here are examples from Reddit [7] and AWS [8]). Google has simply been running dynamic control systems for longer than most organizations. With the rise of SDN, service meshes, job orchestration, and autoscaling, many more of us are now working with dynamic control systems—and it's important that we understand their drawbacks as well as their many advantages.

References

[1] S. Mandal, "Lessons Learned from B4, Google's SDN WAN," presentation slides, USENIX ATC '15: http://bit.ly/atc15-mandal.

[2] C. Perrow, *Normal Accidents* (Princeton University Press, 1999), p. 277.

[3] Google Compute Engine Incident #16007: https://status.cloud.google.com/incident/compute/16007.

[4] M. Suriar, "Anycast Is Not Load Balancing," presentation slides, SREcon17 Europe: http://bit.ly/srecon17-europe-suriar-slides.

[5] R. I. Cook, M.D., "How Complex Systems Fail": https://web.mit.edu/2.75/resources/random/How%20Complex%20Systems%20Fail.pdf.

[6] Google Cloud Networking Incident #19009: https://status.cloud.google.com/incident/cloud-networking/19009.

[7] "Why Reddit was down on Aug 11 [2016]": https://www.reddit.com/r/announcements/duplicates/4y0m56/why_reddit_was_down_on_aug_11/.

[8] "Summary of the December 24, 2012 Amazon ELB Service Event in the US-East Region": https://aws.amazon.com/message/680587/.