Dan Woods Follow
Jan 15, 2019 · 6 min read

On Infrastructure at Scale: A Cascading Failure of Distributed Systems

At Target, we run a heterogeneous infrastructure in our datacenters (and many other places), where we have multiple different backend hosting infrastructure for workloads. Most of this is a legacy artifact of putting infrastructure into production for different use-cases and application development and deployment patterns. The Target Application Platform (TAP) provides a common interface for running and managing workloads, where we can spread workloads across different hosting infrastructure transparently to applications. We use this as a tool to better manage capacity, fully utilize the infrastructure we have available, and to move applications from one hosting provider to another without requiring application teams to re-platform.

We have many core services that act as centralized systems for the entire enterprise. We intentionally push this pattern so we can provide highly available systems and services that can be consumed, instead of requiring every team to be an expert on running infrastructure in addition to developing and managing their applications. Examples of these enterprise services are databases, elasticsearch, and messaging.

At Target, we heavily leverage Kafka as a message broker between systems. Applications use Kafka to inform other, disconnected systems about data changes, and for a variety of other reasons. In infrastructure, we use Kafka to ship logs and metrics from applications to the appropriate backend systems. This way, we can decouple the production of insight data from the consumption, storage, and availability of that data. Every application that deploys through TAP is deployed with logging and metric sidecars, which makes it easy for application teams to consume this data from centralized systems without any additional setup time or configuration.

Kafka runs in the datacenter OpenStack infrastructure, and one evening last week an upgrade was performed to OpenStack's underlying network subsystem. This upgrade was meant to cause only a brief disruption to network connectivity. Due to issues encountered during the upgrade, network connectivity was disrupted for several hours, resulting in intermittent connectivity to Kafka.

We run many Kubernetes clusters underneath TAP, however there is one cluster that was built significantly larger than the others and hosted around 2,000 development environment workloads. (This was an artifact of the early days of running "smaller clusters, more of them", when we were trying to figure out the right size of "smaller".) Under normal operations, the logging sidecars for applications consume an insignificant amount of CPU time. When Kafka became intermittently available, this caused the logging sidecars to all "wake up" simultaneously, and although they still weren't using a lot of CPU, it was cumulatively enough to put a high load on the docker daemons for the nodes in the Kubernetes cluster.

The higher load on the docker daemons caused the nodes to report to Kubernetes they were unhealthy, at which time the Kubernetes scheduler attempted to move workloads off the affected nodes and on to nodes which were still healthy. Since the Kubernetes scheduler does not evenly balance workloads across all the nodes in the cluster, some nodes were able to withstand the higher CPU usage while others were not. Upon rescheduling workloads from unhealthy nodes to healthy nodes, the nodes that were previously healthy became unhealthy, and the cycle perpetuated.

In addition to logging and metric sidecars, every workload is deployed with a Consul agent sidecar, which allows applications across disparate infrastructure backends to participate in the broader compute environment. As soon as a workload is started, the Consul agent registers it as a service within Consul and begins gossiping out the node's membership to the Consul gossip mesh.

During the rescheduling event, Kubernetes produced approximately 41,000 new nodes, which were quickly spun up and terminated. Although the applications inside the pods were never started before they were rescheduled, the pods were online long enough to register themselves with Consul and message out to the gossip mesh. Registration of the new nodes to Consul resulted in higher CPU usage on Consul, which added intolerable latency to requests to Consul. Furthermore, the increase in number of nodes in the gossip mesh also resulted in overall higher CPU usage for all deployed workloads (compounding the cumulative effect from before).

Consul agents only process a number of messages from other nodes during any given loop. This meant that messages sent to the gossip mesh indicating that nodes had gone away were not processed by peer nodes. Overall, this created a "wave" effect within the gossip mesh, where phantom nodes would expire and be subsequently re-added to the mesh.

During the gossip poisoning event, several other systems became affected. The development environment Vault cluster, which serves secrets to applications, sealed itself since it couldn't communicate with Consul in a reasonable amount of time. TAP's deployment engine, which communicates with Consul to discover its infrastructure backends, cut tokens for applications, and dynamically configure ephemeral load balancing was also affected. Because it couldn't communicate with Consul within a tolerable amount of time, deployments began failing.

After days of research, debugging, and trying one thing after another, we were able to recover Consul and Vault by enabling gossip encryption on Consul. This meant that any messages from the poisoned gossip mesh were rejected immediately. After that, we were able to stabilize the deployment engine and we initiated a full redeployment of the development environment to push out the changes to encryption. As the environment was redeployed, instances not configured for encryption were terminated.

Afterwards, we did some research on Consul and found that in version 1.2.3 there was a PR merged for working with large clusters. We performed an in-place upgrade to Consul, which we were able to measure the success of by performing a controlled scale-up of an individual workload, and subsequent un-graceful scale down. The nodes were quickly removed from both the gossip mesh and the service catalog.

(Side note: workloads from TAP are registered in Consul in the form of "application/cluster", where "application" and "cluster" correspond to the Spinnaker terminology. We do this so we can enforce Consul ACLs at an application level. There was a bug in the Consul 1.2.3 UI that prevented clicking through on a service that has a "/" in the name. I'm not ashamed to admit that I use this UI every single day. Upgrading to Consul 1.3.0 fixed that bug, so that's where we landed in the end.)

Luckily, this problem only affected the TAP development environment, and it could've been significantly worse than it ended up being. However, from the TAP team's perspective, there is an intense focus on making sure TAP provides a world class developer experience without disruption. We think about the development environment as our "prod0", and so this outage was addressed with the highest level of severity.

This problem did not affect the TAP production environment in the same way due to it simply being a much smaller environment. The same net effect was felt in production ("prod1"), however because the Kubernetes clusters there were more sparsely packed, the cascading failure did not occur. We confirmed later on that the gossip mesh in production was also indeed poisoned, however the load was significantly smaller due to smaller number of nodes, and upgrading Consul quickly remediated that.

A couple of things I take away from this outage:

Top highlight

1. My mantra "smaller clusters, more of them" is affirmed. The workloads we had in the smaller development Kubernetes clusters were not affected the same as the big one. Same goes for prod. And we would have been hosed if dev and prod were on the same cluster.
2. Shared Docker daemon is a brittle failure point. Need to work harder to reduce the surface area of an outage to an individual daemon. This goes in hand with "smaller clusters, more of them".
3. I've had concerns about the sidecars in the past, however after this event I am convinced that having each workload ship with their own logging and metric sidecars is the right thing. If these had been a shared resource to the cluster, the problem almost certainly would have been exacerbated and much harder to fix.

Building platforms and designing infrastructure for Target's scale is a unique and incredibly interesting problem. If you're up to it and want to join the team, check out the Careers page or drop me a line at dan [dot] woods [at] target [dot] com.

Docker Kubernetes Kafka Platform

WRITTEN BY
Dan Woods
Follow