a year ago

Outage post-mortem

Fix

From January 18 to January 22, Moonlight's website and API had intermittent outages. These started as occasional API errors but culminated in periods of downtime yesterday. The problems have been resolved, and the application has returned to normal.

Background

Kubernetes is the application hosting software that Moonlight uses. Kubernetes runs applications on groups of servers. The servers are called **nodes**. Copies of an application that run on the node are called **pods**. Kubernetes has a **scheduler** that dynamically decides which pods should run on which nodes.

Timeline

The original errors on Friday appeared to be connectivity problems with a Redis database. Moonlight's API uses Redis on every authenticated request to validate sessions. We noticed that our Kubernetes monitoring reported that some nodes and pods were unresponsive. Our hosting provider Google Cloud reported networking service disruptions, and we assumed that the outage was the root cause.

As traffic declined over the weekend, the errors seemed to resolve with only occasional problems. On Tuesday morning, the Moonlight website was offline with zero external traffic reaching the cluster. Finding another person on Twitter with similar symptoms led us to believe that Google's hosting was experiencing a networking outage. We contacted Google Cloud Support, who quickly escalated the outage to their support engineering team.

Google's support team identified a pattern in the nodes of our managed Kubernetes cluster. When the nodes experienced periods of sustained 100% CPU usage, the virtual machine would experience a kernel panic and crash.

Cause

The cycle that caused the outage appeared to be:

1. Kubernetes scheduler assigned multiple pods with high CPU use to the same node
2. The pods consumed 100% of the CPU resources on the shared node
3. The node experienced a kernel panic, resulting in a period of downtime where the node appeared unresponsive to the scheduler
4. The scheduler would reassign all of the crashed pods to a new node, which repeated the same process - and compounded the effects

Initially, only the Redis pod was experiencing this crashing error. But, eventually, all pods serving web traffic were going offline, resulting in a complete outage. Exponential back-off rules resulted in longer periods of downtime.

Fix

We were able to restore website functionality by adding anti-affinity rules to all of our major deployments. This spreads the pods out across nodes automatically, which increases failure tolerance and improves performance.

Kubernetes is designed to be a fault-tolerant hosting system. Moonlight runs three different server nodes for fault tolerance, and we run three copies of every web-facing application. The intention was to have one copy on each node, which would allow two node failures before downtime. However, Kubernetes occasionally scheduled all three website pods on the same node, which created a single point of failure. Other CPU-intense applications (specifically - server-side rendering) were being scheduled onto the same node, rather than separate ones.

A properly-functioning Kubernetes cluster should be able to manage sustained periods of high CPU usage and move pods around to utilize available resources efficiently. We are continuing to work with Google Cloud support to identify and fix the root cause of the server kernel panics.

Conclusion

Anti-affinity rules make web-facing Kubernetes applications more fault-tolerant. If you run a user-facing service on Kubernetes, you should consider adding them.

We are continuing to work with Google to identify and fix the cause of the node kernel panics.