Aaron Pollack
February 16th, 2017
Napster

Too Big To Failover:

Lessons Learned Scaling Redis in Production

Redis is one of many tools Napster employs to deliver a high-quality streaming experience to our millions of users. Today we enjoy some of the best features of Redis, including lightning fast lookup time (~10ms) and redundancy that allows our on-call engineers to sleep soundly at night knowing that a failed master node will be failed over in 30 seconds. The path to a Redis service that we could rely on was not without bumps, however. In this paper I describe the pains that our API team faced growing a production Redis system and how pitfalls can be avoided for those looking to integrate Redis in their own stack.

In the beginning: Redis optimized exclusively for speed

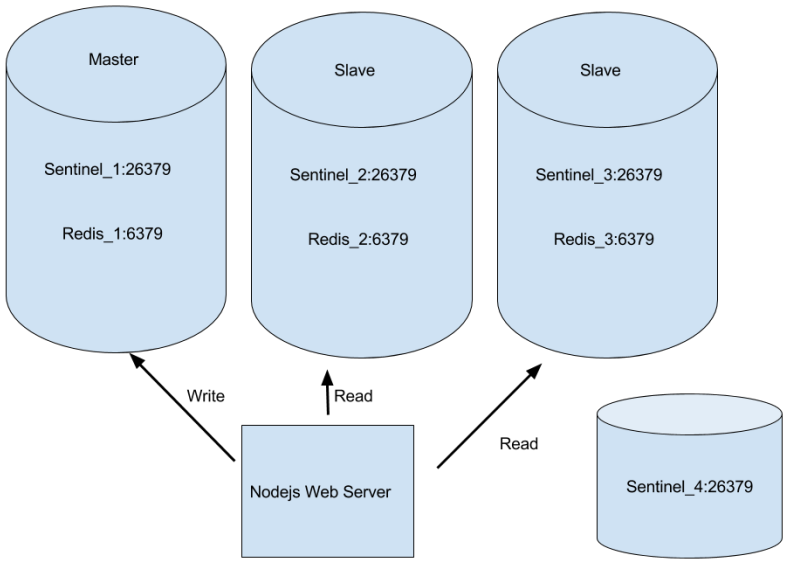

Figure 1. Architecture (Optimized for Performance)

The job of our team is to translate and expose backend data to third-party developers and internal developers. We have used Redis since 2014 to store auth tokens and developer credentials. When I joined the Napster API team (NAPI), I knew very little about Redis other than that it was a new memory store that was gaining popularity. A system was chosen that promised fast lookup time to ensure that we were not the bottleneck for our users' requests. We

were seeing about 4k/s reads to 2 slave nodes and 500/s writes to 1 master node. We also have 4 sentinels for automatic failover. This was the Redis world as I knew it:

Figure 1. Original Speed-optimized Redis Architecture (2014)

This system was architected for speed. There were 10 times the number of reads to writes, so we were able to split the load of reading among two nodes and only send the writes to master. This system works well until a master becomes unreachable; then we start seeing problems.

Main problem observed:

Failover is built to make a master available immediately, but is not designed to keep the slaves available. When a failover is initiated, the slaves will drop in memory data and sync from the new source of truth (the new master). This process makes the slaves unable to serve traffic. As the data scales and the slaves serve traffic under load, the syncing and rehydration of memory to the slaves scales linearly.

Other problems with this architecture:

1. The Redis server shares the same host as the sentinel.
    a. The loss of one host threatens the availability of the data and the failover mechanism.
    b. This limits the number of open ports that the sentinel and Redis server can use. We saturated the default 28K linux limit.
2. An even number of sentinels leads can lead to two sets of sentinels each electing their own master or quorum unreachable.
    a. We never actually experienced this problem firsthand, but this number of sentinels is discouraged in documentation.

When load testing Redis, you have to consider two variables — the growth of the data size, as well as the volume of traffic that it is receiving. Finally, what happens in the worst case scenario? With high traffic and a small data set, the total time of failover is fairly inconsequential, but as the data grows, the time that a slave is unreachable grows as well. Here is an outline of what will happen during failover:

1. Master is unreachable
2. Sentinels reach quorum and failover is initiated
3. A new slave is elected master
4. Master serves traffic
5. New master does full BGSAVE
6. Master syncs data to existing slaves
7. Data is loaded into memory
8. Slave serves traffic

Data in Memory	1GB	5GB	20GB	40GB
Sentinels Detect Failover and Reach Quorum (seconds)	30	30	30	30
Master Does Full BGSAVE (seconds)	9	49	181	243
Slaves Sync to Master (seconds)	39	122	305	425
Data is loaded in Slave Memory (seconds)	8	43	238	354
Total Time (minutes)	*1.5*	*4*	*12.5*	*17.5*

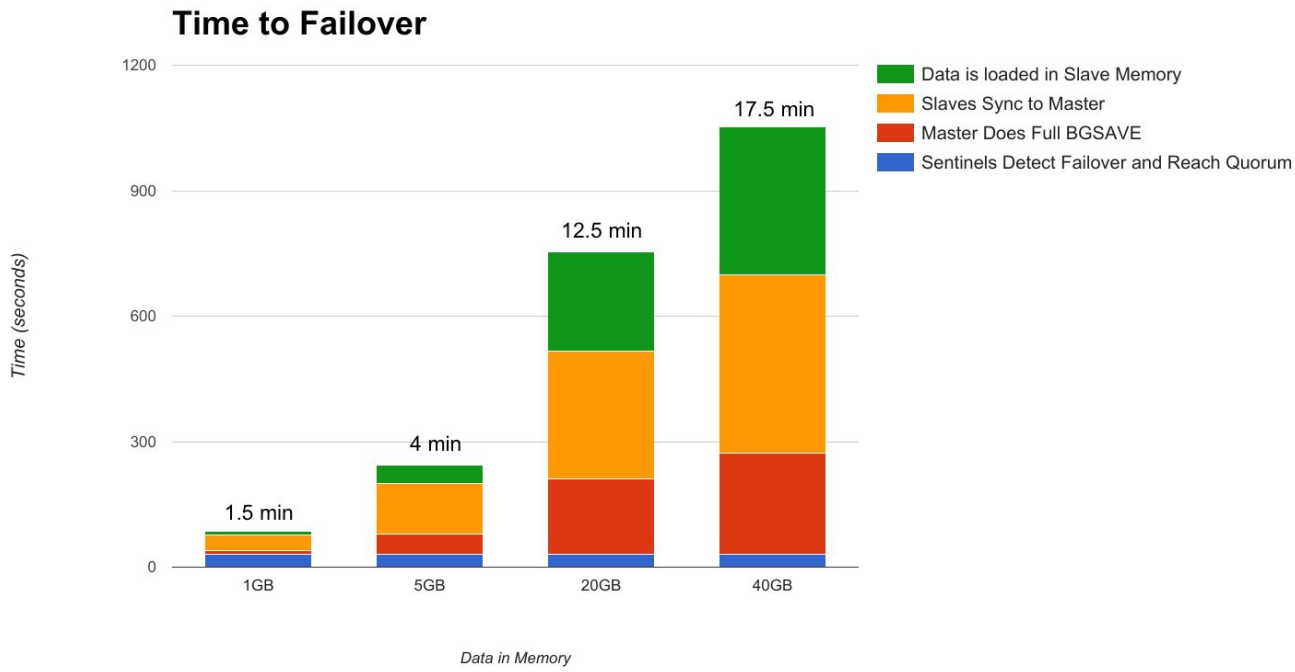

Figure 2. Failover Time as Data Set Size Increases (Optimized for Speed)

Given that we relied on reading from a slave, we found that as our data set grew from bringing on new clients, the downtime we would experience during a failover grew as well. To make matters worse, clients will immediately continue to retry the request upon failure. This meant that the number of connections to each box grew, requiring memory and CPU from the OS to deal with this overhead.

Lessons Learned

Connection Pooling: TCP creation is expensive and it costs your web app and the Redis server to deal with these new connections. If you are creating them faster than you can cull (or let timeout naturally), you make a bad problem worse. We also found that limiting each node process to one connection to Redis actually improved performance on our network.

All Traffic Goes to Master: You lose high availability when using Master/Slave replication during a failover. Failover only works when all connections are to master. If you need the speed of multiple Redis clusters you need to shard your data and have multiple masters.

Think About Data As Volatile and Nonvolatile: Because Redis is so flexible, it's easy to overload it with responsibility. It can work as a cache to read data quickly, a cache to write data more quickly or a persistence layer where the data needs to be saved and replicated. When we started categorizing our data into different buckets, were able to leverage features of Redis differently to suit the data better. To prevent high i/o on our network storage drive, we disabled BGSAVE for our "transient" data and started treating it more like a semi permanent cache, thus freeing us to use an LRU expiration policy to help make our system more resilient to fast-growing memory. Our persisted data set is much smaller, so we can give the box more resources for regular BGSAVES and keep the default buffer limit during a failover.

Resulting Architecture

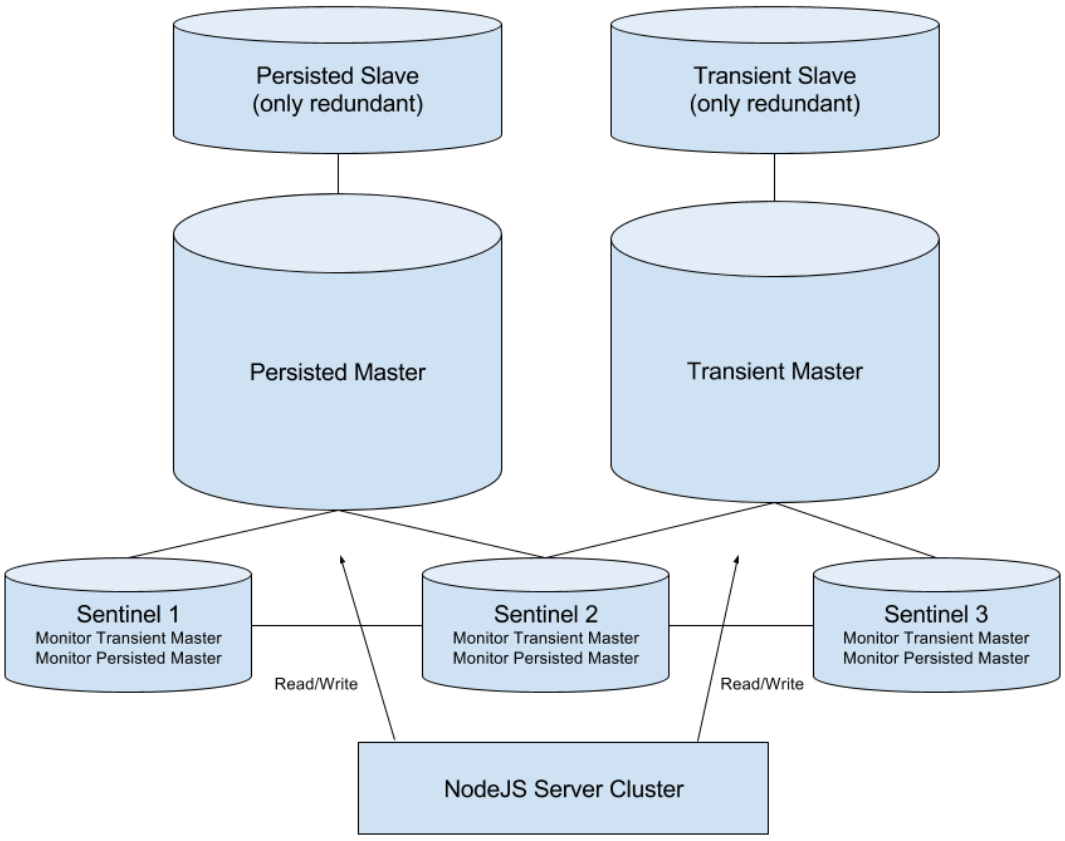

Figure 3. Updated Architecture (Optimized for Availability)

The resulting architecture means that during a failover, we are waiting at most 30 seconds (the default master timeout ping) for the sentinels to agree that a master is down and elect a new one. We achieve failover in constant time regardless of data growth or increases in traffic volume. This has been tremendously helpful for when we have needed to upgrade the version of Redis or the operating system of the host. Additionally we found the the performance degradation from reading and writing to a single master to be modest. At 5 times our peak traffic volume scaled down to a single VM running redis we could write 265,747 authentication token sets in 10 minutes vs 273,565 with the old architecture. A mere 3% performance decrease for true high-availability was an easy trade to make. We love Redis at Napster and consider it an indispensible tool for helping our product grow.