Jintian Liang
CS244B
December 13, 2017

FAWN as a Service

1 Introduction

FAWN, an acronym for "Fast Array of Wimpy Nodes", is a distributed cluster of inexpensive nodes designed to give users a view of a performant key-value store. Through consistent hashing and a log-structured filesystem, FAWN is able to achieve good throughput, capable of satisfying tens of thousands of queries per second [1]. Figure 1 shows the actual configuration of a FAWN cluster. While FAWN is cheap, performant, and energy-efficient, the need to physically assemble such a cluster is an inconvenience that not many are willing to take. Also, because the hardware used to build FAWN nodes is intentionally low-powered, the durability of the nodes' components is questionable meaning that someone has to be tasked with the job of monitoring and replacing failing equipment. Unfortunately, since FAWN utilizes many such nodes to create its cluster, there is a lot of failing equipment to monitor.

Figure 1: Actual FAWN Cluster

Because of these reasons, I wanted to see if it was possible to run FAWN on a cloud service. This addresses many of the issues raised above. For starters, someone who wants to use FAWN's design won't have to purchase the hardware themselves. Many cloud services (such as Amazon EC2 and Google Compute Engine) allow users to customize their instances to their liking. In addition, many of these services often provide monitoring services, sometimes even going as far as to swapping out the failing components without the end user noticing. Finally, an added benefit of emulating a FAWN configuration using a cloud service is that teardown is very simple. There's no need to recycle the parts, which is difficult in FAWN's case since the type of hardware is intentionally underpowered and hard to reapply elsewhere.

I've built a FAWN cluster using nodes from Google Compute Engine. Specifically, I have configured 8 (1 front-end, 7 storage) nodes, all in the same datacenter to reduce network latency during communication. All the nodes are **n1-standard-1** instances, with 1 virtual CPU, 3.75 GB of memory, and 10 GB solid state drives. The virtual CPU is a hardware hyper-thread on a hardware CPU that's at least 2.0 GHz [2]. While the processor is more powerful than those used on the FAWN clusters, the rest of the nodes match up to the specifications of those in the original paper.

2 Implementation

I'm assuming the reader has read the FAWN paper [1], so I'll only be covering how my FAWN implementation differs from the design laid out in the original paper. With that said, there were parts that are integral to the original FAWN implementation that I did not reimplement due to time constraints. More specifically, I did not implement the `Merge` and `Split` routines that are necessary when nodes leave and enter the configuration, respectively. My approach is more of a benchmarking approach to evaluate how performant `Store` and `Lookup` are when performed on a FAWN cluster in the cloud in normal situations, so those two routines are less necessary here. My FAWN implementation was written in Python 2.7 in roughly 400 lines of code.

2.1 Consistent Hashing

FAWN uses consistent hashing in the event that `Merges` and `Splits` occur. When nodes enter or leave a system, it could potentially be expensive to move some objects from many of the nodes to the new node (during a merge) or redistribute objects from the leaving node to many other nodes. The coordination necessary for this would result in a performance bottleneck. Consistent hashing only requires a node's neighbor to be notified of the configuration change, since objects will only be moving to the neighbors. Although my implementation assumes the normal case in which no nodes join or leave the configuration and thus having no need for consistent hashing, it wouldn't be a fair comparison if I didn't try to emulate FAWN to the best of my abilities.

Unlike the 160-bit keys used by FAWN, I used a 128-bit MD5 hash as the key for a key-value object. For consistent hashing, I used the least significant 32 bits as the identifier that determines which node on the ring the key-value gets assigned to. After a `Store` or `Lookup` is routed to the correct node, the least significant 16 bits are used as an index to the node's hashtable that stores the offset for objects in the log file. There are 2^{16} buckets in the hashtable, with each bucket containing the pointer to the beginning of the linked list chain in the event of collision. Each entry in the hashtable contains a 15 bit key fragment that represent the next least significant 15 bits as a means to save memory usage. The final verification of key equality results in reading the entire key in the log-structured filesystem entry.

2.2 Log-Structured Filesystem

One of the reasons why FAWN performs so well especially for writes is its use of a log-structured filesystem. When a value gets updated, the updated value is written to the end of the log file and the in-memory hashtable entry is updated so that the entry's offset reflects the value's new position in the log file. This is powerful in that no synchronization needs to be done when performing I/O on the log file, although some synchronization is needed to modify the in-memory hashtable. The fact that the log file maintains a write-once policy means it is impossible for reads to clash with writes, since all modifications write to the end of the file. Assuming a log-structured filesystem, no synchronization on writes is necessary since the filesystem performs appends atomically. Additionally, append-only operations ensure that no offsets are provided when performing a write, so there's no possibility of writing to a region in the log file that already consists of data. In the original paper, FAWN uses their custom FAWN-DS as their filesystem.

One of the challenges of atomic appends is the synchronization necessary to ensure that the writes are actually atomic. Fortunately, on POSIX-compliant systems, writes that are at most **PIPE_BUF** bytes are guaranteed to be atomic. For Linux systems, **PIPE_BUF** is 4096 bytes [3]. Because FAWN was only evaluated with operations of 256 bytes and 1KB, this is fine for this experiment. However, we still need to be able to calculate the offset at which the new object was written. The problem stems from the fact that there might be multiple clients using the same FAWN cluster, so multiple write requests can occur at the same time. As a result, it isn't guaranteed that the write will occur at the last end-of-file, since a concurrent write can intercept. However, it would be expensive to take a lock on the log file during the append. In fact, one of the defining contributions of FAWN is that it doesn't need to synchronize the log file. It turns out that it's not absolutely necessary to lock the log file if one is willing to pay the cost in file reads. My approach was to take note of the end-of-file position before performing the append. After performing the append, the new entry is guaranteed to be between the last end-of-file and the new end-of-file positions. One can scan this region for the key that was just written and note that position as the offset of the value. This benefits from the synchronization-less read property noted above. Thus, even though writes become a bit more expensive, we maintain the same invariant that no locks are needed when performing I/O.

After performing an append, the head of the replication chain sends a replication command to the next node in the chain. For the purposes of the experiment, each value is replicated once.

3 Evaluation

The original authors of FAWN weren’t very clear in their evaluation methodology in the original paper for FAWN. As a result, I made some assumptions when performing my own benchmarking. I assumed only one client making requests to the front-end server and waiting for the response to come back before sending the next request. Because this is a direct comparison with FAWN, I used the same message sizes for reads and writes, 256 bytes and 1KB. The values are generated randomly.

The results aren’t very promising. Writing 4000 256 byte values for a total of 1MB results in a duration of 148 seconds. Reading the same number of bytes takes 144 seconds. While it’s expected that small writes will take the longest to perform because of the increased overhead costs due to more iterations, this type of performance is unacceptable for any use case. As one would expect, performance improves as you increase the size of messages. Writing 1000 1KB values takes 37 seconds, and reading those values takes about the same amount of time. While this is more respectable than the numbers for 256 bytes, this is still way below the performance for any respectable system. These results pale in comparison to the original FAWN implementation. In contrast, FAWN can perform 40,000 queries per second of 256 bytes and 25,000 queries per second of 1KB. For 256 byte values, my results were three orders of magnitude slower, while they were two orders of magnitude slower for 1KB values.

Note that writes and reads took the same amount of time. This brings up the possibility that the bottleneck for the system lies in overhead costs, as both store and lookup queries go through similar codepaths. In addition, the ratio in duration for 256 byte values compared to 1KB values is the same as the ratio of messages. Reducing the number of total messages sent by a factor of four also reduced runtime by a factor of four. This also means that I/O isn’t the bottleneck, since increasing the number of bytes written from 256 bytes to 1KB per append doesn’t incur additional performance costs. This makes sense given my single-client benchmark and the assumptions that I made earlier. Again, this disparity in numbers stems from the vagueness of the original authors’ evaluation methods. While they provided the specific benchmarks that they ran, they didn’t mention how many clients they were using and whether the number of queries per second is an aggregate over all the clients. As such, it’s impossible to actually replicate the original authors’ evaluation techniques.

With that said, part of the slowness might be a result of increased network latency. Nodes in the original FAWN configuration were connected to one another, meaning they had direct access in terms of communication. While the Google Compute nodes that I used were in the same warehouse, it’s impossible to know how close the nodes are physically without visiting the datacenter. Additionally, the configuration is very sensitive to the network. While testing, I moved the front-end server from the same region as all the other nodes (*us-west-1a*) to another region (*us-central-1a)*. This resulted in a 2x slowdown for 256-byte writes.

4 Conclusion

Because of the abysmal performance I was able to achieve with running a FAWN cluster on the cloud using similarly underpowered nodes as in the original paper, I'm forced to conclude that configuring a FAWN cluster through a cloud service is not advisable. This is further evidenced by the tests that I ran that showed the configuration was sensitive to network effects, which is more prevalent in datacenters than they are in a cluster of physically connected nodes. With that said, this doesn't take away from the original paper, as I do think they had some good ideas, proven by their good performance numbers.

5 References

[1] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. 2011. FAWN: a fast array of wimpy nodes. Commun. ACM 54, 7 (July 2011), 101-109. DOI: https://doi-org.stanford.idm.oclc.org/10.1145/1965724.1965747

[2] *Google Compute Engine Machine Types*. https://cloud.google.com/compute/docs/machine-types

[3] *Linux Manual - Pipe*. http://man7.org/linux/man-pages/man7/pipe.7.html