TaiKV: A Distributed KV Store

Yutai Luo
yutailuo@stanford.edu

Abstract

This paper introduces TaiKV, a scalable distributed KV store. It can be scaled-out easily by adding storage nodes as well as offering high availability by replicating data to multiple machines. Further, multiple improvements have been made to the TaiKV to achieve load balancing.

Introduction

A distributed key-value store, like Dynamo [1] and Cassandra [2], plays a critical role in the large systems nowadays. It is highly partitionable and can be scaled-out easily compared to relational Databases. TaiKV is inspired by Dynamo and Cassandra and offers the following properties:

- Decentralization: a P2P system by leveraging Chord [3]. There are no master nodes and all nodes are equally important. This is to avoid a single point of failure.
- Load Balancing: TaiKV uses consistent hashing [4] to assign data to storage nodes. Consistent hashing reduces the amount of data to be moved in the dynamic environment where node joins and departures are frequent. TaiKV is also able to mitigate hotspots by adding storage nodes to certain key ranges.
- High Availability: TaiKV can be configured to replicate data across different machines. When receiving a read request, it will read from all the replicas in case some of them are not available.
- Eventual Consistency: Data will be stored in all the replicas eventually.

API

The TaiKV API is the following three methods:

- *Get(key []byte)*: this method returns a value corresponding to the key. An error will be returned if the key is not found.
- *Insert(key, payload []byte)*: this method inserts a (key, payload) pair into the TaiKV.

- *Delete(key []byte)*: this method deletes a value corresponding to the key. An error will be returned if the key is not found.

System Architecture

Partitioning

One of the most important properties of TaiKV is highly partionable and can be scaled-out easily. This requires adding storage nodes to serve increasing data and partitioning these data over the nodes in the system. Traditional hashing mechanism (e.g., hash[key] % N, where N is the number of nodes) doesn't work well in the dynamic environment where adding and removing nodes (also called churn) are frequent. In such cases, almost the entire keys need to be remapped. Instead, TaiKV uses consistent hashing to partition data. In consistent hashing, each node is assigned to a circular space. A node is only responsible for the keys fallen between its counterclockwise predecessor and itself. When adding or removing a node, only O(M/N) keys need to be remapped on average where M is the number of keys and N is the number of nodes.

Load Balancing

In consistent hashing, nodes are distributed unevenly in the circular space. This will cause hotspots if certain nodes are responsible for larger key ranges. To improve it, TaiKV supports creating multiple *virtual nodes* per physical node. Thus, one machine can represent multiple points in the circular space and serve multiple key ranges. Another advantage of using virtual nodes is to leverage the heterogeneity in the performance of nodes. We can assign more virtual nodes to a machine if it has better performance.

Although TaiKV has the above improvement, some nodes might become hotspots if certain key ranges serve more read and write requests. For example, a music player service using a KV store as the backend storage, its popular songs will receive more read and write requests which are likely to make corresponding storage nodes become hotspots. To mitigate this case, TaiKV supports assigning a node to a fixed position in the circular space, which acts as splitting the hotspot into multiple nodes.

Replication

TaiKV uses replications to achieve high availability. Each node replicates its data in the following *N* successors, where *N* is a system configurable number. For incoming write requests, each node will store the data into its local persistent storage and then start asynchronous tasks for replication. A reading request will be sent to all its related replicas, the first replica finding the data will return it to the client.

Lookup Protocol

It is critical for a P2P system to look up nodes hosting the data item identified by the key given there are no master nodes to maintain global metadata. A brute-force approach is to start from a random node and look up its successor recursively. This takes O(N) time, where N is the number of nodes in the cluster, which is intolerant for large-scale systems. TaiKV uses the Chord protocol for efficient lookups. In Chord, each node maintains metadata about its predecessor, successor and a finger table. The finger table is the key to efficient lookups. The i^{th} entry in the table at node n is the successor of key $(n + 2^{i-1})$, i.e., $finger[i] = successor(n + 2^{i-1})$. By using the Chord protocol, the lookup time is reduced to O(logN).

To achieve high availability, each node in the TaiKV keeps metadata about its *m* successors, where *m* is a system configurable number. This avoids the failure of its successor to prevent lookups from proceeding.

Node Joins or Departures

Node joins and departures will affect the predecessor, successor and finger table of existing nodes in the system. To be aware of these changes, each node runs periodic background tasks to check its metadata and update it if there is a change.

Failure Detection

Each node in the TaiKV pings its predecessor and successor periodically for failure detection. If certain nodes are not responding, they will be removed from the metadata of their neighbors.

Implementation

Communication Protocol

Communication between TaiKV nodes is implemented by gRPC and Protocol Buffers. The API between the client and TaiKV also uses gRPC and Protocol Buffers.

Local Persistent Storage

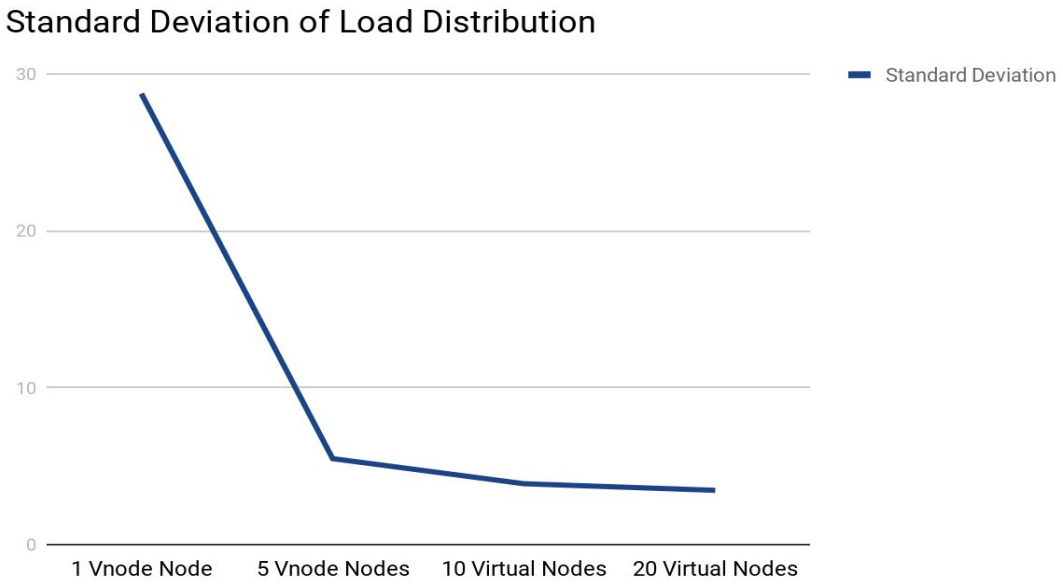
Each node in the TaiKV persists the data into its local disk for durability. Since TaiKV is implemented using Go, it uses BadgerDB [5] as its local persistent storage. BadgerDB is an embeddable, persistent and fast key-value (KV) database written in pure Go. TaiKV is designed to be agnostic to local storages. It can support other types of local storage as well.

Evaluation

We made improvements to distribute loads in the TaiKV. The following benchmark is run with 3 physical nodes in the system. The load distribution with the number of virtual nodes is:

# of Virtual Nodes	Machine A Load Percentage	Machine B Load Percentage	Machine C Load Percentage
1	73.9%	10.9%	15.2%
5	35.5%	38.7%	25.8%
10	37.1%	28%	34.9%
20	31.3%	30.5%	38.2%

The standard deviation of the load distribution is shown as:



As we can see, the load distribution becomes much more evenly after increasing the number of virtual nodes.

Conclusion

This paper describes the design and implementation of the TaiKV, a scalable distributed KV store. There are several improvements that can be made to the system: 1) Using Merkle trees [6] for replica synchronization, Merkle trees can help to detect changed data efficiently and

avoid unnecessary data move; 2) We can further make improvements for load balancing, like distribute traffic across replicas according their loads, etc.

References

[1] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In Proc. 21st ACM Symposium on Operating Systems Principles (SOSP), Oct. 2007.

[2] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44, 2 (April 2010), 35–40. DOI:https://doi.org/10.1145/1773912.1773922

[3] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. SIGCOMM Comput. Commun. Rev. 31, 4 (October 2001), 149–160. DOI:https://doi.org/10.1145/964723.383071

[4] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In Proceedings of the twenty-ninth annual ACM symposium on Theory of computing (STOC '97). Association for Computing Machinery, New York, NY, USA, 654–663. DOI:https://doi.org/10.1145/258533.258660

[5] BadgerDB: https://godoc.org/github.com/dgraph-io/badger

[6] Ralph C. Merkle. 1987. A Digital Signature Based on a Conventional Encryption Function. In A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology (CRYPTO '87). Springer-Verlag, Berlin, Heidelberg, 369–378.