Pythonamo: Dynamo inspired Key-Value Store

https://github.com/vamc19/python-dynamo

Caleb & Vamsi
CS7610 - Final Project
December 2, 2018

Abstract:

This paper documents the design and implementation of the Pythonamo Key-Value Store which is heavily inspired by Amazon's DynamoDB paper and Facebook's Cassandra paper. The actual system which was implemented is not quite either, but borrows techniques from both to achieve a highly available, low latency, key-value store which does not compromise write availability for read latency and is also persistent in the face of transient failures of the hardware or network. The result is a system with configurable availability and durability based on the desired behavior for the underlying client.

Section 1. Introduction:

This project began as an attempt to implement as much of Dynamo's proposed features as we could in a month's time, however; as the implementation grew, many design changes had to be made and adopted from Facebook's Cassandra system (which itself is based upon Dynamo) in order to create a working system in the time provided. These changes stemmed from both opaque wording in the original paper as well as changes for simplicity and efficiency which were made by one or both inspiring systems.

Section 2. Background:

Key-Value Stores and Distributed Hash tables have been an important topic of research and discussion for many years. These systems are fundamental in meeting the scalability and availability requirements of today's commercial and industrial distributed systems. In order to create a usable system within the month, several assumptions were made which simplified the system description as provided in the Dynamo paper.

Section 2.1 Assumptions:

The assumptions made while implementing Pythonamo are as listed in this section. All put requests are forwarded through the master to give them a total ordering. This is also the case for how Cassandra works. Nodes do not need to support virtualization and as a result they are deterministically placed in the topology. There is sufficient time to allow the system to stabilize before permanently adding or removing another node from the ring. There is not any system in place to install a new leader. Cassandra uses Zookeeper and it is not shared how Dynamo does this.

Section 2.2 Availability of Data Vs. Latency:

Like Dynamo, Pythonamo tries to balance constant and pervasive failure with high availability and low latency. The way Pythonamo accomplishes this is by assuming a temporary node or network failure whenever a node does not respond fast enough, to avoid waiting on this faulty node, Pythonamo will replicate the necessary data on other nodes to maintain its safety properties and increase availability while also putting a bound on client latency.

Section 3. Previous Work:

Section 3.1 Dynamo:

Dynamo was started at Amazon to meet reliability requirements at the their scale. It is designed to be highly-available, tolerating network partitions at the cost of consistency. The system only guarantees the data to be eventually consistent. Data stored in the Dynamo is versioned using Vector Clocks and relies on client application to resolve conflicts between different versions.

Section 3.2 Cassandra:

Development of Apache Cassandra began at Facebook to power their Inbox Search feature. Cassandra is very much inspired from Dynamo - it is also designed to be highly available and eventually consistent. Cassandra, however, makes a fair share of design decisions that are different from Dynamo. This can be attributed to the reason that unlike Dynamo, which is designed to run on Amazon's infrastructure, Cassandra aims to run on commodity hardware in any datacenter. Following sections discusses the differences between the designs in more detail.

Section 4. System Architecture:

This section breaks Pythonamo into its separate components and explains how they function from a high level.

Section 4.1 Client Interface:

The client has the ability to store values in the system under a given key with a put command. Values are appended to a key by providing the context from a previous read call or a blank context json object {}. The client can also retrieve the list of values for a given key with a get command. The values for the given key are returned sorted by the context values stored in the system with the corresponding versions of the key-value.

Section 4.2 Partitioning Algorithm: (Vamsi)

Both Cassandra and Dynamo use Consistent Hashing for data partitioning. In this algorithm, each node in the network is placed on a virtual ring. Based on the hash value of the key, one of

the nodes on the ring is made responsible for the data. The data on each node is replicated on to N-1 adjacent nodes in the ring, where N is the replication factor.

We also use Consistent Hashing algorithm to partition and replicate our data. But, our implementation of the algorithm is very rudimentary in comparison Cassandra's and Dynamo's. Our implementation places the nodes on the ring deterministically based on the host name of the node whereas Cassandra and Dynamo use a ring topology to make sure data is being replicated onto the nodes that are physically as far away from each other as possible. Cassandra and Dynamo use Virtual Nodes to load balance the data on each of the nodes. Our implementation does not support Virtual Nodes.

Section 4.3 Replication:

When puts and gets are performed in the system, the coordinator of the request uses the sloppy quorum algorithm to perform the store or retrieve operation on the nodes in charge of replicating the data. The system is configured to have N replicas of the data stored on unique nodes. Put and Get commands require W and R nodes respectively to respond within the timeout for a given request for an operation to be considered successful. These values can be configured for a variety of client applications use to supply the durability and availability necessary for the data.

Section 4.4 Put & Get Methods:

When a get command is sent to a node in the system, if the target node for that key is up, the request is forwarded to that node to be the coordinator of the request, otherwise, the node that received the command will serve as the coordinator. The coordinator contacts all the nodes responsible for replicating the key and asks them to retrieve their version of the key and send it back. When the coordinator has R unique responses to the request, he can forward the correct answer back to the client. If too much time has elapsed before a get request can be completed, the node will respond back to the client with an error.

When a put command is sent to a node in the system, if it is not the leader it sends the put command to the leader. The leader will then order the request and send it to the target node to coordinate the request. If the target node is unavailable, the leader will attempt to coordinate the request itself by contacting the other replicas for the key. When the coordinator has W confirmed writes for the value, it can forward the success message back to the client. If the coordinator cannot write the data to enough nodes within the allotted time for the request, it will respond back to the client with a failure.

Section 4.5 Data Versioning:

When a value is stored into the database, the vector clock count for the number of times a particular coordinator handled a request for that key is increased by one. This can be thought of

as a sparse vector clock array for the servers in the system writing to the shared data variable. In response to a get command, all unique versions of the key are returned sorted by their vector clock values in newest to oldest order and versions that are concurrent are adjacent. The vector clocks are returned as JSON objects with keys for servers and the corresponding values are the number of times they wrote to the database when that value was created.

Section 4.6 Hinted Handoff:

The nodes in the ring are assumed to be unavailable temporarily from time to time due to any number of reasons including network partitions or software failure. During such outages, the data for an unavailable node (A) has to be handed off to a available node (B). B stores the data locally and waits for A to be available. As soon as A comes online, B hands over the data to A and deletes its local copy. It has to be noted that the handoff is not counted towards the successful write requirement (W) and cannot be read. It is only required to make sure that each key value pair is stored at least on N nodes.

For simplicity, a few assumptions were made while implementing hinted handoffs. Only one handoff will be made for each key value pair. This is because we determine the node for hand off deterministically. To provide multiple hand offs, we will have to determine multiple nodes that are currently available and, given that we can only test on a limited set of nodes, data may end up on one of the replicas violating the contract for having data on N different nodes.

Section 4.7 Membership & Failure Detection:

In Dynamo and Cassandra, gossip based protocols are used to propagate membership changes, providing eventually consistency. Dynamo paper does not talk about the gossip propagation or the time it takes for membership to become consistent. Cassandra, on the other hand, does not allow membership changes until the membership awareness spreads to the entire cluster, which is dependent on the size of the cluster and network conditions. Our implementation uses a 2 phase commit protocol for membership changes. This enables our membership to be strongly consistent and completed deterministically.

For failure detection, Both Dyanmo and Cassandra use gossip based protocols. In these systems each node stores locally a list of failures it learnt from gossips and uses this information to perform hinted handoffs. It has to be noted that while the failures are detected by the nodes, they do not provide global view of failures. In our implementation, we detect failures during requests. Each request is expected to be completed within a specific time window. If the request fails or no response is heard from the node within the window, the node is considered to be failed the request is forwarded to a handoff or a replica.

Section 4.8 Permanent Failures:

One of the core assumptions of Dynamo is that permanent failures are very rare and will be handled manually. The data in the nodes are assumed to be saved to a safe storage area (infrastructure dependent, like Amazon's S3) that can be recovered across failures. Our implementation makes sure that all the critical data is saved to disk and is restored if available.

Our implementation supports manual node addition and node removal. Data synchronization should be handled by the tree reconciliation algorithm, which is not implemented lacking time. So, while we support basic infrastructure for it, our implementation cannot handle permanent failures.