Review - Dynamo: Amazon's Highly Available Key-Value Store

Name: Kunal Baweja
UNI: kb2896
COMS E6998 Cloud Computing & Big Data (Fall 2016) October 8, 2016

Introduction

Dynamo[1] is a highly scalable, reliable and distributed key-value storage system for large scale applications, particularly by Amazon.com. It manages state of services that require high reliability and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance.

Dynamo[1] combines a number of techniques to achieve scalability and availability:

- Data is partitioned and replicated using consistent hashing.
- Consistency is provided by object versioning and during updates its provided by a quorum-like technique and a decentralized replica synchronization protocol.
- A gossip based distributed failure detection and membership protocol for reliability.

Motivation

The main motivation for *Dynamo*[1] data storage system is the importance of state management by an application requiring scalability and reliability. *Dynamo*[1] is specifically designed for applications which require high availiability of data store, along with scalability properties and require only primary-key access to a data store. The specialized feature of providing only a key-value storage implementation stems from the fact that many application store and retrieve data by primary key and do not require the complexity of a traditional RDBMS.

System Assumptions and Requirements

Dynamo[1] is based on the following assumptions and requirements for a data storage system:

1. A query model is either a simple read or write operation uniquely identified by a key.
2. An operation does not span multiple data items.
3. State of a data item is stored as binary object identified by a unique key.
4. Unlike standard databases providing ACID (Atomicity, Consistency, Isolation, Durability) properties, *Dynamo*[1] concentrates only on applications with weaker consistency, to rather provide high availability and instead of providing isolation guarantees it allows only single key updates in a single operation.
5. Efficiency is guaranteed by providing a choice for applications to configure *Dynamo*[1] to achieve required latency and throughput via balancing tradeoffs between performance, cost efficiency, availability and durability guarantees. This configuration is further used to negotiate a Service Level Agreement(SLA) with the client requesting a service, to provide a response in bounded time.

The following sections provide a concise summary of the architecture and design of *Dynamo*[1] key-value data store.

Design Considerations

Consistency: *Dynamo*[1] is designed to be an eventually consistent data store, with replication for high availability, meaning that once it receives a data update it propogates it in background to the replicas of the data, however there might be conflicts in view of the data at a given time instance between replicas which is gracefully handled by *Dynamo*[1] via certain conflict resolving mechanisms.

Conflict Resolution: Traditional data stores, which execute conflict resolution during writes and reject writes if not all/majority of replicas of data can be accessed at the given time. However, *Dynamo*[1] is meant to be an "always writeable" data store, hence rejecting writes is not a viable option, especially for applications targeting customers over the web. So *Dynamo*[1] handles any conflicts in data view at time of data read operations. *Dynamo*[1] allows the applications using it to specify their own conflic resolution mechanisms.

Incremental Scalability: *Dynamo*[1] can scale out one storage node at a time with minimal impact on system as well as operator.

Symmetry: Every node in *Dynamo*[1] cluster is treated equally, no concept of master-slave nodes.

Decentralization: The design favours peer-to -peer techniques to provide high availability and scalability, including the gossip protocol for failure detection.

Heterogeneity: *Dynamo*[1] also tries to exploit heterogenity of the physical infrastructure, i.e. work distribution is proportional to the capabilities of individual storage nodes.

System Architecture/Implementation Details

The core distributed systems techniques used in *Dynamo*[1] are: partitioning, replication, versioning, membership, failure handling and scaling. Each of these are described individually as:

1. **System Interface**
   Dynamo[1] has a simple interface that provides get() and put() operations to retrieve and store/update data respectively. The put() operation takes key, context and object arguments, determines where to put the replicas of the object based on the MD5 hash of the key and writes the replicas to the disk. The get() operation takes key as an argument and retrieves an data object along with its context. Here, context refers to the encoded system metadata about the object which includes information such as version number etc.

2. **Partitioning Algorithm**
   Dynamo's partitioning scheme relies on consistent hashing to distribute load across multiple storage hosts i.e partioning of data stored. In consistent hashing the output range of the hash function is treated as a "ring" and each node in the system is assigned a random value within this ring, identified as the "position" of the node. Each data item's key is hashed and the item is assigned to a node by matching the node's "position" to the hash value.

   In the matching mechanism, once the hash is calculated on key of data item, its destination node's position is determined by scanning through the circular space of assigned positions of nodes and once a node with position value greater than calculated hash is found, the data is

stored in that node. The advantage of this approach is that in case a node goes offline, the next available node in the circular space takes over in its place.

3. **Replication**
   Dynamo[1] replicates its data on multiple hosts to achieve high availability and ruability. Each key is assigned to a co-ordinator node which is responsible for duplication of data. The co-ordinator node stores the key within itself and the N-1 successor nodes in the storage node ring, where number N is configured by application using *Dynamo*[1].

4. **Data Versioning**
   Dynamo[1] provides eventual consistency which allows updates to propagated to replicas asynchronously. Hence a put() call may only update the data in some of the nodes/copies of data. These updates are guaranteed to propogate in background to all the replicas, however sometimes due to component failure this might not happen and may result in subsequent get() calls getting an inconsistent data view, which is resolved by the resolution mechanism specifed by the application.

   Further, to guarantee "always writeable" property, even in the case of unavailability of the latest data version, *Dynamo*[1] treats each modification as a new and immutable version of data. Hence multiple versions of the data can be present in the system at the given time and most of the times newer version subsumes the older version, except in case of conflicts, where a version branching may happen due to application of update operation on an older version, which is eventually resolved at the time of read. Versioning is taken care of by using vector clocks, wherein a list of *(node, counter)* pair is used to identify the different versions of an object.

5. **get() and put() operations**
   Any storage node in *Dynamo*[1] can receive client requests for operations which can be routed to the desired node in two ways:

   - Through a generic load balancer that will select a node based on load information.
   - Using a partition-aware client library that routes requests directly to appropriate co-ordinator node.

6. **Handling Temporary Failures**
   To avoid failures, *Dynamo*[1] performs read and write operations on the first N healthy nodes from the preference list, and these N healthy nodes might not be the same as the first N nodes from the consistent hashing ring. Along with this a "sloppy quorum technique" is followed wherein the application using *Dynamo*[1] can specify the minimum number of nodes to which data should be written or read from for acceptance of the read/write operations.

7. **Handling Permanent Failures**
   Dynamo[1] implements a replica synchronization protocol using Merkel Trees to keep the replicas synchronized which allows faster detection of inconsistencies between replicas along within minimal data transfer as the comparisons are made on the hash values stored within these Merkle Trees.

8. **Membership and Failure Detection**
   In *Dynamo's* storage nodes consistent hash ring an explicit mechanism is required to add or remove a node from the ring, which is handled via an administrator who issues the relevant commands. These commands about membership changes are recorded at the node to which

the command is issued and communicated to other nodes in the ring via a gossip based protocol which eventually maintains a consistent view of membership of nodes.

Failure detection is implemented internally in *Dynamo*[1] to avoid attempting communications with a failed node. A given node A determines another node B to be failed if B does not responds to A's communincation request within a bounded time period. In such case node A contacts alternate nodes which map to the same partition as B to service the requests. This failure detection is also based on gossip protocol where in the nodes randomly ping each other at fixed intervals.

9. **Adding/Removing Storage Nodes**
   A new node added into the *Dynamo*[1] system is assigned a number of tokens which denote a random hash mapping on the consistent hash ring, described above. For a new node added to the system, a range of keys are assigned to it for storage and since some of those keys would have been previously assigned for handling to the previously present nodes, they will now stop recording the data items for those keys and transfer the keys to the newly added node.

   The exact reverse happens in removal of a storage node, where the keys that were assigned to the node removed are now assigned to the nodes left within the *Dynamo*[1] system, based on the hash range those keys belong to in the consistent hash ring.

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 *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, (New York, NY, USA), pp. 205–220, ACM, 2007.