Dynamo: Amazon's Highly Available Key-value Store

G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels

Alexandre Milouchev, MIT (6.897 Spring 2011)

1 Introduction

At a massive scale, there is no question whether failure will occur or not – it surely will, and it will do so often. To Amazon.com, with its fleet of services, reliability is one of the key concerns that need to be addressed in order to provide an "always-on" experience to its clients. To that end, Amazon's core services use Dynamo, which is a highly-available, eventually consistent, key-value distributed storage system.

2 Design

In order to achieve the high level of availability it provides, Dynamo by design sacrifices consistency to some extent. It uses the principle of eventual consistency, which is a specific form of weak consistency. The latter provides no guarantee that after a write is performed, a read will return the updated value. With eventual consistency, it is certain that at some point after a write, provided no other writes occur, all readers will see the same updated value. Inconsistency must be tolerated by a very-large-scale reliable distributed system, in order to (1) improve the speed at which large volumes of reads and writes are performed, and (2) to ensure availability in network partition cases where requiring majority (and therefore enforcing strong consistency) would render part of the system temporarily unavailable.

In Dynamo's case, the design decision to use eventual consistency came from the conclusion that many of Amazon's services can handle temporary inconsistency, in exchange of increased performance and, most importantly, availability. The paper also states that the majority of these services don.t need a full-fledged RDBMS to operate, which would incur significant increases in maintenance costs and reduced performance, and that a key-value type of storage would be sufficient. It is important to note that the design of Dynamo was facilitated by the assumption that it will only be used by Amazon internal services, so the requirements of the clients were known in advance; moreover, no security concerns were raised during the design phase. The other main design question raised by the paper is on conflict resolution. Conflicts are guaranteed to occur with optimistic replication techniques. Since strong consistency is not enforced, the system will often hold multiple replicas of the same data that are "out of sync". When an update is first executed, it needs to be propagated to the other nodes that store replicas of the same data. Conflict resolution can be handled either by the server or by the client, and either at read or at write time. Typically, distributed systems handle conflict resolution at write time, so that read complexity is reduced. However, that can lead to cases where a write is rejected and needs to be repeated. In the case of Dynamo, that is not an acceptable scenario, as many of the services that run on top of it need to be able to perform updates on the data no matter what (e.g.: amazon.com shopping cart update cannot be lost). That is why conflict resolution is migrated to the read operations, thus allowing Dynamo to be an "always writeable" data store.

3 System architecture

The paper mentions that besides dealing with availability and data persistence, such a massively distributed system needs to have scalable solutions for a number of other concerns/subcomponents. Six of those are specifically addressed in the system architecture section.

1. Partitioning

The network will necessarily be partitioned for a system of this scope. To deal with that partitioning, Dynamo uses consistent hashing. When a new node is added to the system, it gets assigned a number of virtual nodes on a hashing "ring". Each virtual node deals with a range of values from the output space of the hash function in use. This is different from the general DHT idea, where each node has a single position on the ring. The advantages of using virtual nodes are threefold: the burden of sending replicas to a newly-added node is shared by multiple nodes; the load handled by a node that fails is spread across multiple of the remaining live nodes; finally, a new node can be assigned a variable number of virtual nodes, depending on its capacity, thus taking into account different physical nodes. discrepancies in performance.

2. Replication

To ensure both durability and availability, data is replicated not only on the ring node within whose hash range its hash falls, but also on the N subsequent clockwise nodes. The list of nodes responsible for a given key is called its "preference list". A preference list is longer than N nodes to account for failures. Finally, a key's preference list is guaranteed to contain different physical hosts by purposefully skipping virtual nodes which are hosted on a physical host that has already been used, at construction time.

3. Data Versioning

Due to eventual consistency, a write operation might not take effect immediately on all nodes that hold copies of the data. Thus there will be out-of-sync versions of the data in the system. In order to detect causality between them, Dynamo uses vector clocks. In this way, if during a read it realizes that there are multiple versions of the same data in the system, it can check for causality and return the later version if there is a causal relation. If there isn't, however, Dynamo cannot infer anything and will simply return all of the different existing replicas. Conflict resolution occurs if the client then proceeds to make an update to the data – it is assumed that the client has then reviewed the received information and reconciled it into one.

4. Put(key, context, object) and Get(key), eventual consistency

Dynamo uses a quorum-like approach to reads and writes. One node acts as the coordinator for the operation. There are two predetermined values, R and W, which are used to gather the necessary number of nodes for a read or for a write, respectively. The coordinator starts by contacting the top N reachable nodes from a key's preference list. In case of a read, if R nodes respond with the data, it checks for causally unrelated versions and returns all of them. For a write, the coordinator waits to hear back from at least W-1 nodes before considering the operation a success.

5. Permanent failure: Replica synchronization

In order not to lose unique replicas due to permanent failure of a node, Dynamo uses Merkle trees to enforce replica synchronization. Each node holds a Merkle tree for each range it handles. In that way, two nodes dealing with the same range can very quickly compare the top of their Merkle tree, then decide whether they need to synchronize or not.

6. Membership

Since nodes can fail at any time then quickly recover, there needs to be a mechanism for signaling when a node is added to the system for the first time, or when it is about to be removed permanently intentionally. In such a case, the node itself makes writes to permanent store about its membership change, and then lets that change propagate through the ring using a gossip protocol. That protocol has the dual purpose of also propagating virtual node placement information. In that way, every node knows about the other nodes. ranges and can forward put or get requests in constant time.

4 Implementation and production optimizations

In its implementation, Dynamo has three components present in each node, all of which are implemented in Java: request coordination, membership, and failure detection. A node also has a persistence engine which can be setup to use one of a number of different storage engines. The two most popular ones are MySQL and Berkeley Database Transactional Data Store, with the first being the default options for bigger objects, and the latter being used when handling small objects (10-100KB).

In production, the point where Dynamo really shines is its adaptability to various use cases. For conflict resolution, for example, Dynamo lets the client choose if reconciliation will be handled by the client or the server. Many business applications choose to handle reconciliation themselves, but a few perform well with simple timestamp reconciliation on the server side, where if there are multiple versions of the same object at read time, the one with the most recent timestamp is returned. The other feature that makes Dynamo very versatile is the fact that it exposes N, W, and R to the client. In that way, one can optimize the storage for reads (W=N, R=1), or for quick writes (R=N, W=1). The most typical configuration used in production is (N, R, W) = (3, 2, 2).

5 Conclusion

The last part of the paper demonstrates, with production data, how Dynamo has been successful so far in providing the type of highly-available storage that Amazon's services require. It comes to show that an eventually-consistent system, built using a variety of decentralized algorithms, can successfully handle the real-life requirements of services used in what is currently one of the most challenging application environments.