# Dynamo - Amazon's Highly Available Key-value Store

Prakhar Srivastav (ps2894)

Dynamo came out of a need to provide a datastore for those services that have very high reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance.

In particular, the following were the target requirements that Dynamo is designed to handle -

1. Dynamo is targeted mainly at applications that need an "always writeable" data store where no updates are rejected due to failures or concurrent writes.
2. Dynamo is built for an infrastructure within a single administrative domain where all nodes are assumed to be trusted.
3. Applications that use Dynamo do not require support for hierarchical namespaces or complex relational schema.
4. Dynamo is built for latency sensitive applications that require at least 99.9% of read and write operations to be performed within a few hundred milliseconds.

To meet these target requirements, the designers of Dynamo embraced the following design principles -

- Incremental scalability: Ability to scale out one storage host at a time, with minimal impact.
- Symmetry: Every node in Dynamo should have the same set of responsibilities as its peers.
- Decentralization: An extension of symmetry, the design should favor decentralized peer-to-peer techniques over centralized control.
- Heterogeneity: Work distribution to be proportional to the capabilities of the individual servers.

The key idea behind Dynamo is to give services control over their system properties, such as durability and consistency, and to let services make their own tradeoffs between functionality, performance and cost-effectiveness.

## Architecture

**Eventual Consistency in Dynamo**: From the very early replicated database works, it is well known that when dealing with the possibility of network failures, strong consistency and high data availability cannot be achieved simultaneously. For systems prone to server and network failures, availability can be increased by using optimistic replication techniques, where changes are allowed to propagate to replicas in the background, and concurrent, disconnected work is tolerated. Dynamo is designed to be an **eventually consistent datastore**, that is updates reach all the replicas eventually. In order to be an "always writeable" data store, the system pushes complexity of conflict resolution to the reads in order to ensure that writes are never rejected.

**API**: Dynamo is a complex storage system that provides a very simple API involving just two operations - `get` and `put`. The `get(key)` operation locates the object replicas associated with the key in the storage system and returns a single object or a **list of objects with conflicting versions** along with a context. The `put(key, context, object)` operation determines where the replicas of the object should be placed based on the associated key, and writes the replicas to disk.

**Partitioning**: Dynamo's partitioning scheme relies on consistent hashing to distribute the load across multiple storage hosts. In consistent hashing, the output range of a hash function is treated as a fixed circular space or "ring". The principle advantage of consistent hashing is that departure or arrival of a node only affects its immediate neighbors and other nodes remain unaffected. To keep the heterogeneity of the nodes into consideration Dynamo's variant adds virtual nodes into the system.

**Replication**: Like all other distributed storage systems, Dynamo replicates its data on multiple hosts. Each data time is replicated at $N$ nodes and each key $k$ has a respective coordinator node. The coordinator is in charge of the replication of the data items that fall within its range which it then replicates at the *N-1* clockwise successor nodes in the ring. This list of nodes (called *preference list*) is also made available so that every node can determine which nodes should be in this list.

**Data Versioning**: The eventual consistent nature of Dynamo makes it inevitable that some data in the system will be stale. eg. a put followed by a get may not always return the most updated version. Dynamo treats the result of each modification as a new and immutable version of the data. It allows for multiple versions of an object to be present in the system at the same time. In cases where the system cannot reconcile the multiple versions of the same object and the client must perform the reconciliation in order to collapse multiple branches of data evolution back into one (semantic reconciliation). More importantly to deal the possibility of having multiple variations of the same data, Dynamo uses Vector Clocks in order to capture causality between different versions of the same object. Vector clock information is used for reconciliation which happens when the user provides the `context` in the `put` call. Upon processing a read request, if Dynamo has access to multiple branches that cannot be syntactically reconciled, it will return all the objects at the leaves, with the corresponding version information in the context. This means that the divergent versions and the branches are collapsed into a single new version.

**Executing get() and put() operations**: Any storage node in Dynamo is eligible to receive client get and put operations for any key. A node handling the read or write operation is known as the coordinator which generally is the first of N nodes in the preference list for that key. The responsiblity of locating a coordinator can be either from the partition aware client library or from a simple load balancing scheme (but with few hops) to locate the coordinator. Read and write operations involve the first N healthy nodes in the preference list, skipping over those that are down or inaccessible. To maintain consistency among its replicas, Dynamo uses a consistency protocol similar to those used in quorum systems and provides two configurable values - $R$ (the minimum number of nodes that must participate in a successful read operation) and $W$ (the minimum number of nodes that must participate in a successful write operation). Configuring $R$ and W such that $R + W > N$ yields a quorum-like system.

- `Get()`: The coordinator requests all N highest ranked reachable nodes for that key and waits for R responses before returning the results to the client.
- `Put()`: The coordinator generates a vector clock for the new version and writes the new version locally. It then sends this update to the N highest ranked reachable nodes and considers a write successful if at least $W - 1$ respond.

**Transient Failures**: To be highly available, Dynamo does not enforce strict quorum membership and instead it uses a "sloppy quorum"; all read and write operations are performed on the first N healthy nodes from the preference list, which may not always be the first N nodes encountered while walking the consistent hashing ring. The *hinted handoff* is process whereby in which when a node fails the next node is allowed to write the key but is provided with metadata (*hint*) indicating the original node the key was meant for. When the original node comes back up, the new node will attempt to transfer the data back to it and eventually delete it from its local store.

**Permanent Failures**: In the face of permanent failures, a more robust replica synchronization protocol is required. The anti-entropy protocol used by Dynamo relies on Merkle trees which is structured so that each branch of the tree can be checked independently without requiring nodes to download the entire tree or the entire data set. Each node maintains a separate Merkle tree for each key range (the set of keys covered by a virtual node) it hosts. This allows nodes to compare whether the keys within a key range are up-to-date.

**Membership**: To guard against manual errors for unintentional startups and departure of unreachable nodes, it was deemed appropriate to use an explicit mechanism to initiate the addition and removal of nodes from a Dynamo ring. The mechanism is explicity driven manually by an administrator. The node that serves the request writes the membership change and its time of issue to persistent store. This history file is then transmitted to the other nodes in the system using a gossip-based protocol which maintains an eventually consistent view of membership. To prevent logical partitions, some Dynamo nodes play the role of seeds. Seeds are nodes that are discovered via an external mechanism and are known to all nodes. Because all nodes eventually reconcile their membership with a seed, logical partitions are highly unlikely. Failure detection in Dynamo is used to avoid attempts to communicate with unreachable peers during get() and put() operations and when transferring partitions and hinted replicas. Early designs of Dynamo used a decentralized failure detector to maintain a globally consistent view of failure state. Later it was determined that the explicit node join and leave methods obviates the need for a global view of failure state. This is because nodes are notified of permanent node additions and removals by the explicit node join and leave methods and temporary node failures are detected by the individual nodes when they fail to communicate with others (while forwarding requests).

## Lessons Learned

The authors of the paper share a set of interesting lessons that are the key takeaways of this paper

- The main advantage of Dynamo is that its client applications can tune the values of N, R and W to achieve their desired levels of performance, availability and durability. For example, if W is set to 1, the system will never reject a write request as long as there is at least one node in the system that can successfully process a write request. To act as a high performance read engine with infrequent updates, client applications set R to be 1 and W to be equal to N.
- Achieving high performance is hard since the involvement of multiple storage nodes causes the performance of the read and write operations to be limited by the slowest of the R or W replicas. To provide even higher levels of performance, an object buffer was added that kept the writes in memory to be periodically written to storage by a writer thread.
- To keep the load evenly distributed the authors do various modifications to the partitioning scheme. The overall idea of these modification strategies was to decouple the schemes for data partitioning and data placement.
- The number of divergent versions of the same data that are seen by the application in a live environment are good proxies understanding the impact of failures on the system.
- To ensure that background tasks ran only when the regular critical operations are not affected significantly, the background tasks were integrated with an admission control mechanism.
- Instead of keeping the state machine on the server, it was found that moving the state-machine to the client had important advantages one of which was that the load balancer was no longer required. If the membership lists were keep up-to-date this system proved to be very efficient.

## Conclusion

The main contribution of this work for the research community is the evaluation of how different techniques can be combined to provide a single highly-available system. It demonstrates that an eventually-consistent storage system can be used in production with demanding application.