The Life of Dynamo’s put() & get() Operations

Let’s learn how Dynamo handles get() and put() requests.

We'll cover the following

Strategies for choosing the coordinator node#

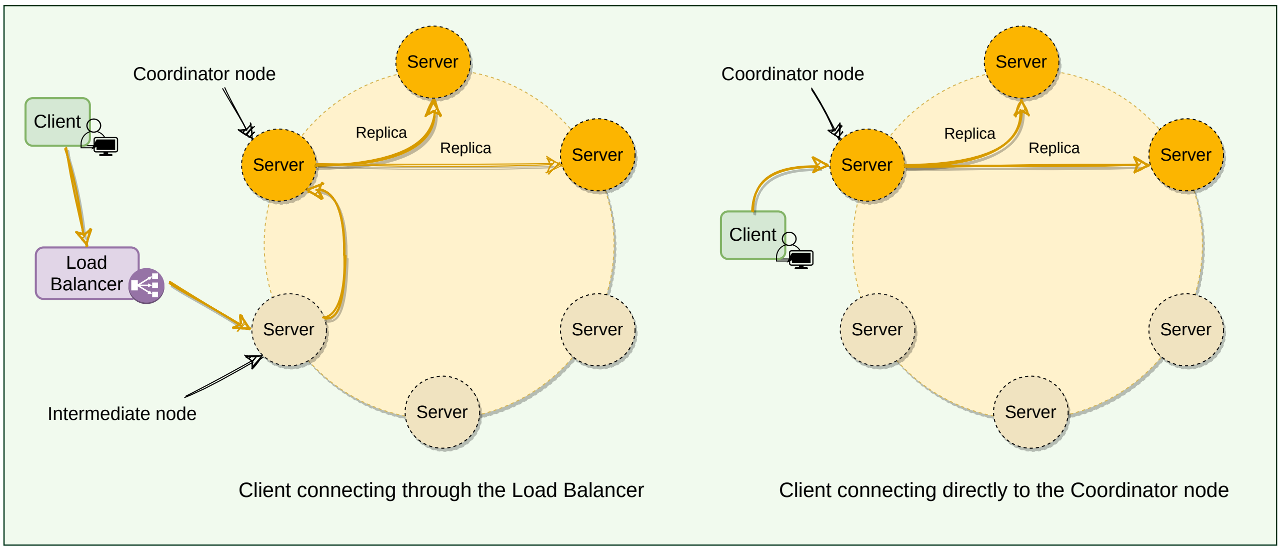
Dynamo clients can use one of the two strategies to choose a node for their `get()` and `put()` requests:

- Clients can route their requests through a generic load balancer.
- Clients can use a partition-aware client library that routes the requests to the appropriate coordinator nodes with lower latency.

In the first case, the load balancer decides which way the request would be routed, while in the second strategy, the client selects the node to contact. Both approaches are beneficial in their own ways.

In the first strategy, the client is unaware of the Dynamo ring, which helps scalability and makes Dynamo’s architecture loosely coupled. However, in this case, since the load balancer can forward the request to any node in the ring, it is possible that the node it selects is not part of the preference list. This will result in an extra hop, as the request will then be forwarded to one of the nodes in the preference list by the intermediate node.

The second strategy helps in achieving lower latency, as in this case, the client maintains a copy of the ring and forwards the request to an appropriate node from the preference list. Because of this option, Dynamo is also called a **zero-hop DHT**, as the client can directly contact the node that holds the required data. However, in this case, Dynamo does not have much control over the load distribution and request handling.



How clients connect to Dynamo

Consistency protocol#

Dynamo uses a consistency protocol similar to quorum systems. If R/W is the minimum number of nodes that must participate in a successful read/write operation respectively:

- Then $R + W > N$ yields a quorum-like system

- A Common (N, R, W) configuration used by Dynamo is (3, 2, 2).
  - (3, 3, 1): fast W, slow R, not very durable
  - (3, 1, 3): fast R, slow W, durable
- In this model, the latency of a `get()` (or `put()`) operation depends upon the slowest of the replicas. For this reason, R and W are usually configured to be less than N to provide better latency.
- In general, low values of W and R increase the risk of inconsistency, as write requests are deemed successful and returned to the clients even if a majority of replicas have not processed them. This also introduces a vulnerability window for durability when a write request is successfully returned to the client even though it has been persisted at only a small number of nodes.
- For both Read and Write operations, the requests are forwarded to the first ‘N’ healthy nodes.

‘put()’ process#

Dynamo’s `put()` request will go through the following steps:

1. The coordinator generates a new data version and vector clock component.
2. Saves new data locally.
3. Sends the write request to $N - 1$ highest-ranked healthy nodes from the preference list.
4. The `put()` operation is considered successful after receiving $W - 1$ confirmation.

‘get()’ process#

Dynamo’s `get()` request will go through the following steps:

1. The coordinator requests the data version from $N - 1$ highest-ranked healthy nodes from the preference list.

2. Waits until $R-1$ replies.
3. Coordinator handles causal data versions through a vector clock.
4. Returns all relevant data versions to the caller.

Request handling through state machine#

Each client request results in creating a state machine on the node that received the client request. The state machine contains all the logic for identifying the nodes responsible for a key, sending the requests, waiting for responses, potentially doing retries, processing the replies, and packaging the response for the client. Each state machine instance handles exactly one client request. For example, a read operation implements the following state machine:

1. Send read requests to the nodes.
2. Wait for the minimum number of required responses.
3. If too few replies were received within a given time limit, fail the request.
4. Otherwise, gather all the data versions and determine the ones to be returned.
5. If versioning is enabled, perform syntactic reconciliation and generate an opaque write context that contains the vector clock that subsumes all the remaining versions.

After the read response has been returned to the caller, the state machine waits for a short period to receive any outstanding responses. If stale versions were returned in any of the responses, the coordinator updates those nodes with the latest version. This process is called **Read Repair** because it repairs replicas that have missed a recent update.

As stated above, `put()` requests are coordinated by one of the top N nodes in the preference list. Although it is always desirable to have the first node among the top N to coordinate the writes, thereby serializing all writes at a single location, this approach has led to uneven load distribution for Dynamo. This is because the request load is not uniformly distributed across objects. To counter this, any of the top N nodes in the preference list is allowed to coordinate the writes. In particular, since each write operation usually follows a read operation, the coordinator for a write operation is chosen to be the node that replied fastest to the previous read operation, which is stored in the request’s context information. This optimization enables Dynamo to pick the node that has the data that was read by the preceding read operation, thereby increasing the chances of getting “**read-your-writes**” consistency.

← Back
Vector Clocks and Conflicting Data

Next →
Anti-entropy Through Merkle Trees

✓ Mark as Completed

Report an Issue