Dynamo: Amazon’s Highly Available Key-value Store

### 2020310795 康鸿博

## What are the problems mentioned by the paper? (intro)

Dynamo is a distributed Key-value store. It’s designers build it to be a “high availability, high performance(low 99.9% latency), scale out” large scale distributed storage system that supports hundreds of independent nodes around the world.

The main challenge Dynamo’s designers faced is that, traditional RDBMS can’t provide the “availability, latency, and scalability” it wants. It’s because traditional RDBMS wants to provide good consistency promises on the semantic level, which is well known as a main hinderance to good performance on a distributed storage system.

However, modern workloads, like the one in Amazon, has shown its internal difference compared with the workload in traditional RDBMS. They’re different in many aspects:

1. They mainly use the primary key to access data. Few operations access multiple primary keys at the same time, and few require a “transaction”.
2. They can tolerate some inconsistency, and the programmers can write codes to solve these problems.
3. They show different requirements on “availability, consistency, and latency”, and want to make tradeoffs for a stable performance.

## Summary of major innovations (intro)

Dynamo is a eventual consistent large-scale k-v store. It uses “consistent hashing” for data partition and replication, “object versioning” for consistency, “quorum-like consensus and decentralized replica synchronization” for replicas management, and gossip-based algorithms for failure detection and membership record.

The main contribution of Dynamo is that it shows how these different algorithms work together as a whole system, and point out several lessons they learnt. The typical contribution of Dynamo is it proves that system frameworks may make compromises on the consistency, and let the programmers to do the fix.

## How about the important related works/papers? (related work)

As a system paper that focus on how algorithms work together, it’s related work is about its components.

Consistent hashing is a well-known algorithm in P2P network. It was first designed to support flood-based message passing, then extended to be a structured distributed data structure. However, they’re mainly designed to be in an untrusted distributed system, and to resolve all consistency issues.

Distributed file systems or databases seems to fit Amazon’s need, but none of them actually do. Some of them use centralized node for better performance, which can be dangerous on failures; some of them provide security guarantees or range operations, which isn’t Amazon’s interest, etc.

## What are some intriguing aspects of the paper? (design & implementation)

###### Interface:

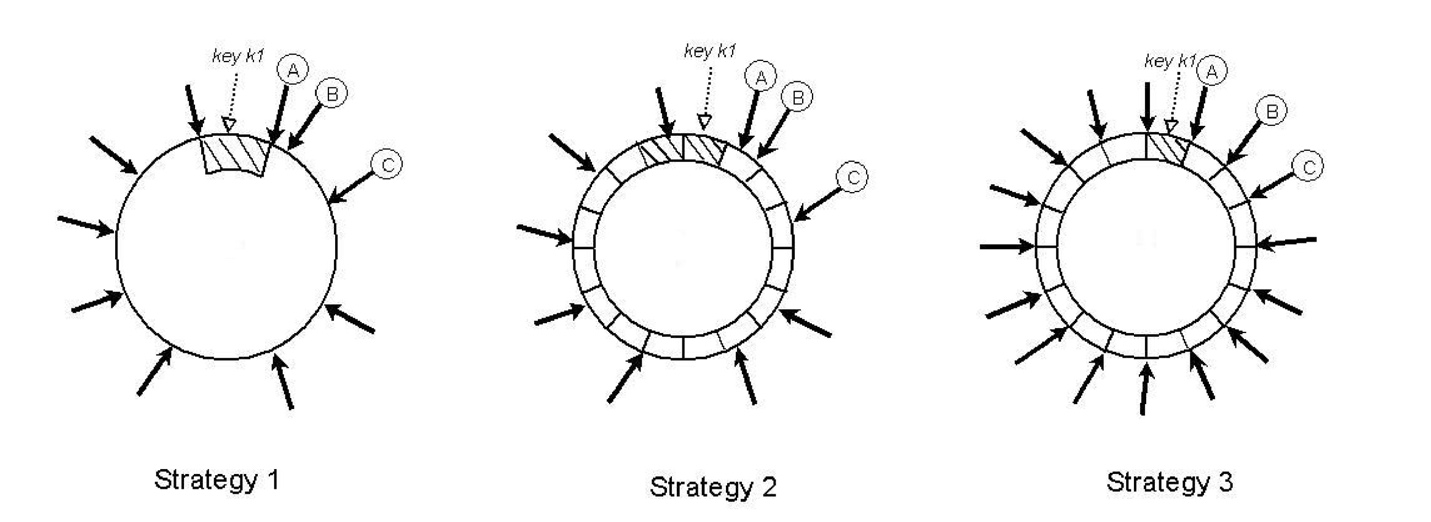
Put(key, context, value) -> true/false

Get(key) -> value, context

###### Consistent Hashing(Figure3):

As in the following figure, consistent hashing is a 2-step algorithm. For each (key, value) pair, it hashes the key into the hash space. Each key should belong to some range in the hash space controlled by one node. It then stores the pair to the that node.

In Dynamo, the total hash space is divided into Q subranges, and each subrange is assigned to some node randomly, which guarantee the load balance.



###### Replication:

To support high availability and durability, Dynamo replaces its data on multiple nodes. To be specific, each (key, value) pair is not only stored on the node holding its range, but also on nodes holding the following ranges. This enable load balance with replication, and support easy data retrieval on failures.

###### Data versioning:

Since Dynamo only support eventual consistency, it can exploit more parallelism by data versioning. Its versioning is similar to that in “causal consistent” systems, where each version is identified by an “update list”, containing timestamps for all relevant nodes. The difference between the consistency algorithm in Dynamo and that in “causal consistency” algorithms is that it allows nodes to reply to “get” requests with a value that may not be the “latest” in the system. This may be a result from node failure, or too many concurrent accesses. This brings the problem that multiple concurrent versions for the same key may exist in the system. Dynamo don’t solve this, but send all these versions, and let the conflict solvers provided by the program engineer solve this.

###### Quorum-based replication management:

For each get/put operation, traditional quorum-based algorithms send it to the N corresponding nodes, and wait until receiving R / W (constants) confirmations. In Dynamo, a weaker quorum is used, where W writing nodes don’t necessarily be in the first N nodes.

This allow “put” operations to succeed even if the “original node” in the quorum fails. The substitute nodes will then send the value back after the node recover.

Note that the substitute node may fail, so the original node will lose the update. Dynamo solves this by Merkle tree.

###### Membership and failure detection:

Membership is maintained by a list. Nodes periodically gossip to each other to synchronize the list (like in many network protocols). To support fast synchronization, some nodes are picked by the administrator as seeds, and all nodes try to sync their state with seeds. Node failures are detected by heartbeat messages. Adding / Removing nodes are done by calling some function.

## How to test/compare/analyze the results? (experiment)

The author report results generated by the execution of Amazon products. It shows that the system reaches good load balance, especially when under high load. It also shows that system designers shouldn’t just focus on the average latency, because the

99.9% latency can be orders of magnitude higher than the average latency. Focusing only on average latency may lead to bad user experience.

## How can the research be improved? (the bad side, future work, your idea)

For my own idea (or future research interest), I’m interested in: (1) how can we apply small “transaction” if needed? (2) how can we utilize the “heterogeneity” in the network? Since message passing in datacenter is much faster than those between datacenters.

## If you write this paper, then how would you do? (your idea)

I like this paper very much, and I don’t think I can do better than it. I think it’s a good system paper, which has brave hypothesis from real world (instead of theoretical analysis), clear paper structure, and long-term real-world data.

## What’s your test Results about the paper? (your action)

Nope.

## Give the survey paper list in the same research area (your survey)

[1] [Fay Chang](https://dblp.org/pid/13/6463.html), [Jeffrey Dean](https://dblp.org/pid/d/JeffreyDean.html), [Sanjay Ghemawat](https://dblp.org/pid/g/SGhemawat.html), [Wilson C. Hsieh](https://dblp.org/pid/31/1764.html), [Deborah A. Wallach](https://dblp.org/pid/53/6138.html), [Michael Burrows](https://dblp.org/pid/73/615.html), [Tushar Chandra](https://dblp.org/pid/26/477.html), [Andrew Fikes](https://dblp.org/pid/71/5497.html), [Robert Gruber](https://dblp.org/pid/g/RobertGruber.html):  
**Bigtable: A Distributed Storage System for Structured Data (Awarded Best Paper!).** [OSDI 2006](https://dblp.org/db/conf/osdi/osdi2006.html#ChangDGHWBCFG06): 205-218

[2] [Venugopalan Ramasubramanian](https://dblp.org/pid/85/4250.html), [Emin Gün Sirer](https://dblp.org/pid/s/EminGunSirer.html):  
**Beehive: O(1) Lookup Performance for Power-Law Query Distributions in Peer-to-Peer Overlays.**[NSDI 2004](https://dblp.org/db/conf/nsdi/nsdi2004.html#RamasubramanianS04): 99-112

[3] [Antony I. T. Rowstron](https://dblp.org/pid/r/AITRowstron.html), [Peter Druschel](https://dblp.org/pid/d/PDruschel.html):  
**Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility.**[SOSP 2001](https://dblp.org/db/conf/sosp/sosp2001.html#RowstronD01): 188-201

[4] Lu, Y., Shu, J., Chen, Y., & Li, T. (2017). Octopus: an rdma-enabled distributed persistent memory file system. In 2017*USENIX Annual Technical Conference*(USENIX ATC 17) (pp. 773-785).