Distributed Systems

Christian J. Rudder

January 2025

Contents

This page is left intentionally blank.

Big thanks to **Prof. Anna Arpaci-Dusseau**
and **Prof. Ioannis Liagouris** for teaching CS351: Distributed Systems
at Boston University [1].

All illustration contain original assets.

Disclaimer: These notes are my personal understanding and interpretation of the course material. They are not officially endorsed by the instructor or the university. Please use them as a supplementary resource and refer to the official course materials for accurate information.

Prerequisites

This text assumes the reader has a basic understanding of computer science and programming. It will also assume they are somewhat familiar with computer architecture and operating systems at a high level. The text will review these concepts briefly for completeness, but it will not try to teach them from scratch or provide a full understanding of these topics.

The main focus will be on distributed systems, and will touch on:

- **Concurrency and Parallelism**
  - Concurrency, Parallelism, Threads
- **Consistency and Fault Tolerance**
  - Consistency, Fault-tolerance, Atomicity
- **Distributed Systems and Coordination**
  - Asynchrony, Coordination, Logical Time, Snapshots
- **Consensus Algorithms**
  - Raft, Paxos, Consensus
- **Replication and Data Management**
  - Replication, Sharding, Cluster
- **Protocols and Computing Models**
  - RPC, 2PC, Broadcast
- **Technologies and Tools**
  - MapReduce, Spanner, Dynamo, GFS, TLA+, Golang

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

It's 2007 and Amazon's global e-commerce platform must remain "always-on" despite continual component failures, outages cost revenue and customer trust. To achieve this, Dynamo sacrifices strict consistency for low-latency availability:

Definition 1.1: Design Goals of Dynamo

Dynamo is a decentralized, highly available key-value store designed to:

- **Always-Writeable:** Never reject writes under partitions or node failures, deferring conflict resolution to the read path.
- **Incremental Scalability:** Scale out by adding or removing nodes without downtime or manual repartitioning.
- **Decentralized Symmetry:** No central coordinator, based on Consistent Hashing (**??**).
- **Low-Latency Performance:** Dynamo provides its clients an SLA (Service Level Agreement) that under any load, it provides a 300ms response 99.9% of the time.
- **Eventual Consistency:** Allow temporary inconsistencies under failures, ensuring all updates reach replicas eventually.

Consistency Model: Weak Consistency, favoring availability over strict consistency.

Next we discuss how it achieves this decentralized, highly available key-value store:

Definition 1.2: Quorum System – Coordinating (Part 1)

A user's keys is hashed to a point on a 128-bit ring and mapped to an ordered list of N nodes (the **preference list**). The first clockwise node, N_i, becomes the **coordinator**:

- **Writes**: Coordinator N_i receives and sends `put(key,value)` in parallel to the next top $N-1$ replicas, waits for acknowledgments from any W distinct replicas (**including itself**), then returns success to the client.
- **Reads**: N_i receives and sends `get(key)` to all $N-1$, returning success after R nodes acknowledge the read.
- **Conflicts**: Divergent data is reconciled via a vector-clock versioning history. If the vector clocks cannot be merged, it is left to the **client** to resolve.
- **Quorum Condition**: Choosing parameters $R + W > N$, ensures R and W overlap, diminishing staleness.
- **Ownership:** If key k hashes to A's segment, A is the **primary** coordinator when it's healthy. If A is unreachable, the **next alive** node temporarily stands in as coordinator.

We'll call the below a turtle-back diagram; it illustrates the quorum system in a basic configuration:

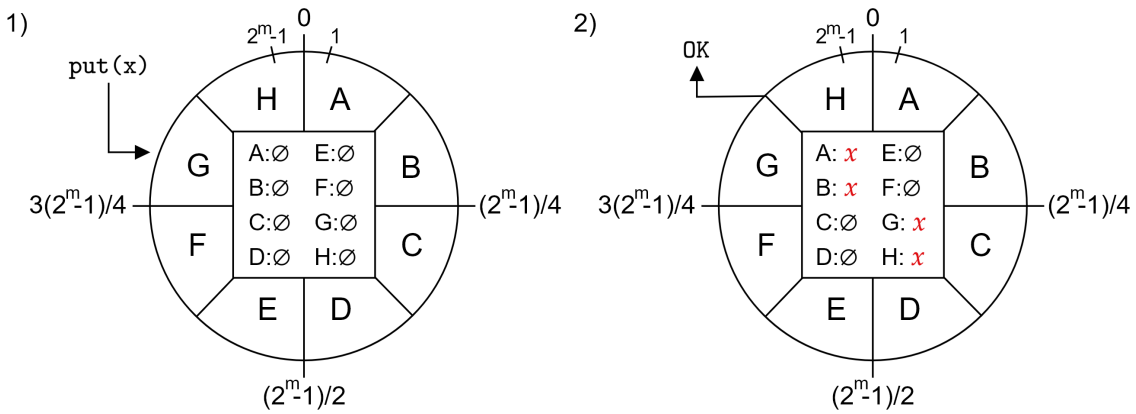


Figure 0.1: A basic quorum system. **To be clear**: virtual nodes are positioned at the spokes. For instance, H starts at 0, and ends at the next left-spoke G (the top-left corner of the center box). Here, $R = 4, W = 4, N = 7$, and servers are an ordered list of virtual nodes A–H. Also, here we say `put(x)` for brevity, while it's actually `put(key,value)`. 1) A client's put request falls within G's range. 2) The next top $W - 1$ nodes acknowledge, with G sending back an `OK` (success).

Moving on, we deal with the liveness of nodes, and how to reconcile data:

Definition 1.3: Quorum System – Gossip & Hints (Part 2)

Dynamo's monitors liveness with the following mechanisms:

Gossip Frequency & Peer Selection: Every second, each node picks a random peer and exchanges its local membership-change log (join/leave/failure records). This gossip ensures that all nodes eventually learn who is up or down without any central service.

Failure Detection & Sloppy Quorum: During a write, if a coordinator cannot reach a preferred replica (due to a failure or partition), it writes to the next healthy node in the preference list. That node stores the update alongside a **"hint"** tagging the intended replica.

In particular, each node stores hints in a separate local database unbeknownst to the client. This is called a **sloppy quorum**: it allows writes and reads to proceed even if some replicas are unreachable, as long as W and R are satisfied.

Hinted Handoff: When the failed node rejoins, any node holding a hint for it will detect its liveness via gossip and then forward the update it missed in a "handoff"—restoring the full set of N replicas **without** blocking client operations.

Below illustrates the gossip protocol and hinted handoff:

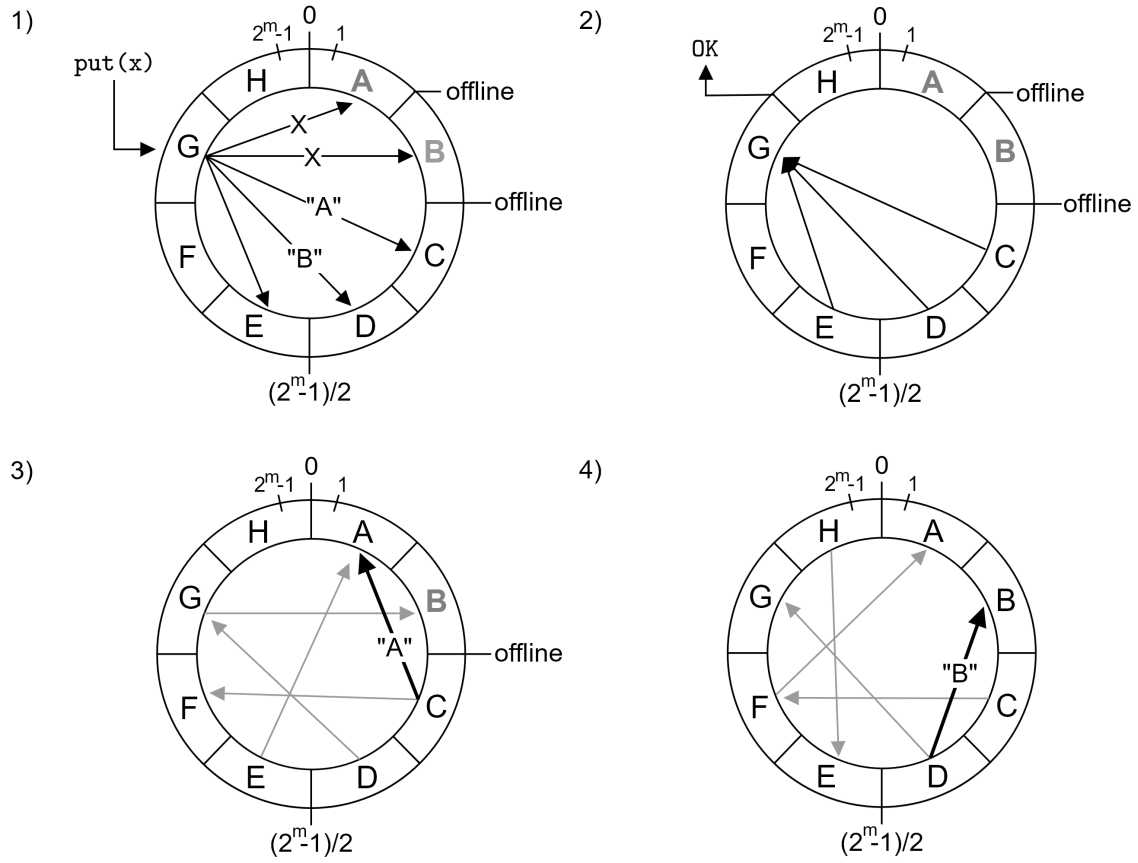


Figure 0.2: Gossip protocol and hinted handoff. 1) A put operation is sent to G, from which is then propagated. Though, A and B appear to be down, G resolves this by giving hints to C and D. 2) Nodes C, D and E `ACK`, allowing G to return `OK`. 3) During the gossip protocol A comes back online; C notices this and sends the hint to A (hinted-handoff). 4) Later in the gossip protocol, B comes back online; D notices and hands off the hint to B. **Note**: The paper does not follow the one-hint-per-replica assumption made in this figure. Also, G **should actually** send the write in parallel to $H \to A \to B \to \ldots$ (its first three successors), but for visual-clarity we began with A.

Note: The Dynamo paper doesn't bound hint capacity. Here we assume at most one hint per missing replica, which in the worst case could lose writes if that holder also fails. We might instead store hints on the next N healthy nodes, or allow multiple distinct hints per replica to improve safety.

The paper does say, "replicas will keep [hints] in a separate local database that is scanned periodically." The use of the word **Database** suggests hints may be propagated generously.

Now we discuss in detail how dynamo utilizes **Vector Clocks** to reconcile divergent data:

Definition 1.4: Quorum System – Vector Clocks & Reconciliation (Part 3)

Every object (key-value pair) in Dynamo is associated with a **vector clock** VC that tracks the version history of the object. The VC is composed of N (node, counter) tuple pairs, maintaining a change history:

- **Stamping & Propagation:** A client invokes $\texttt{put}(k, v, VC_{\text{ctx}})$ at the coordinator N_i, where VC_{ctx} is the VC from the last read ($\texttt{get()}$). N_i merges VC_{ctx}, increments its own counter entry in VC, stores (k, v, VC) locally, and then sends the entire tuple (k, v, VC) in parallel to the other $N - 1$ replicas.
- **Primary Path:** In the failure-free case, N_i waits for W acknowledgments (including itself) before replying success. Each replica simply stores the tuple under key k in its local key-value store.
- **Failover Path:** If N_i is unreachable, the client retries $\texttt{put}(k, v, VC_{\text{ctx}})$ at the next alive node in the preference list. That node becomes the temporary coordinator: it fetches or merges existing clocks, stamps its own counter, performs a sloppy-quorum write of (k, v, VC), and tags hints for any skipped replicas (including the primary).
- **Read Reconciliation:** A client issues $\texttt{get}(k)$ to the coordinator, which polls all N replicas and waits for the first R responses (v_j, VC_j). If one VC_j strictly dominates the others, its v_j is returned; if some VC_j are concurrent, all corresponding v_j are returned.
- **Application Merge:** When multiple versions (v_j, VC_j) arrive, it's up to the application to merge them (semantic reconciliation) into a single v_{merged} and VC_{merged}, then issue a fresh $\texttt{put}(k, v_{\text{merged}}, VC_{\text{merged}})$.

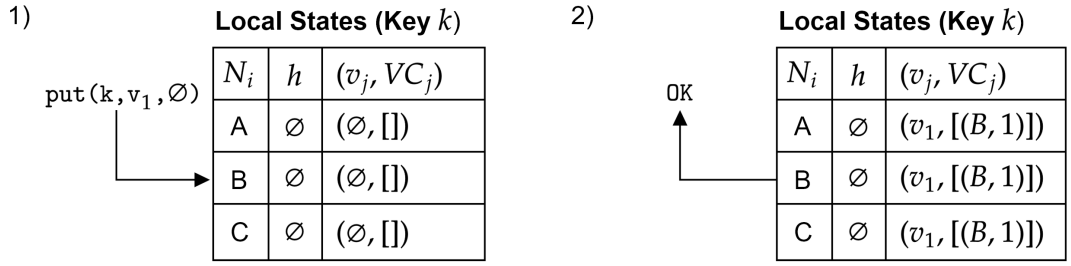
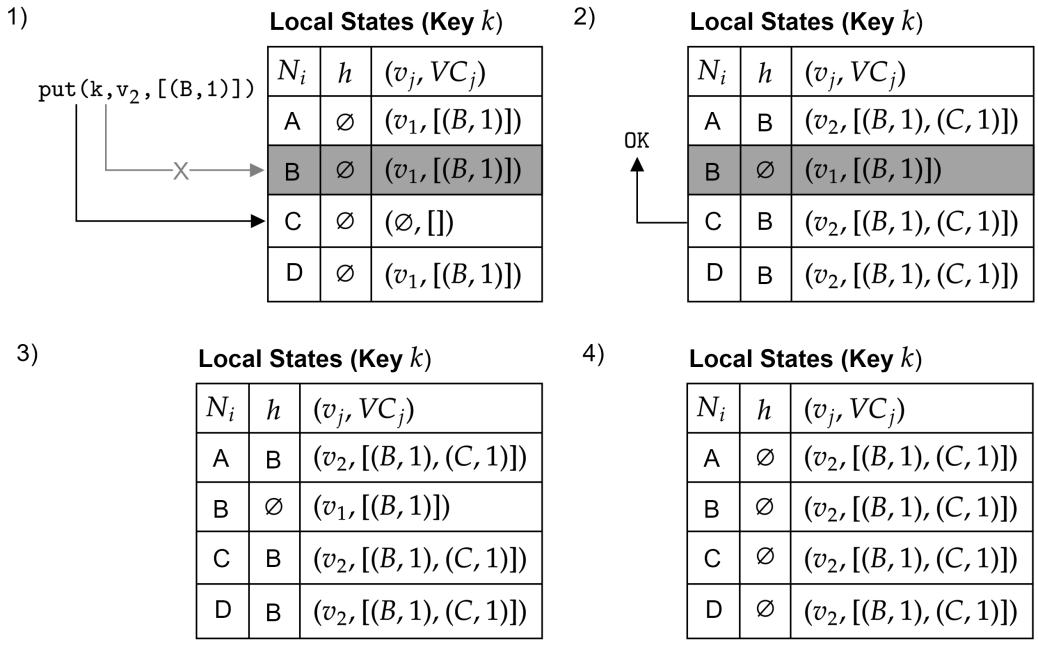


Figure 0.3: A simplified view of a ring with three N_i nodes, A, B, and C, h hints, and a (value, vector clock) tuple (v_j, VC_j). 1) A client issues their first put request of $\texttt{put(k,v}_1\texttt{,}\varnothing\texttt{)}$ (empty context) to B. 2) B stores the tuple, $(v_1, [(B, 1)])$, indicating the new value, and notes its participation in a vector clock VC. It propagates (k, v_1, VC) to A and C, returning success to the client.



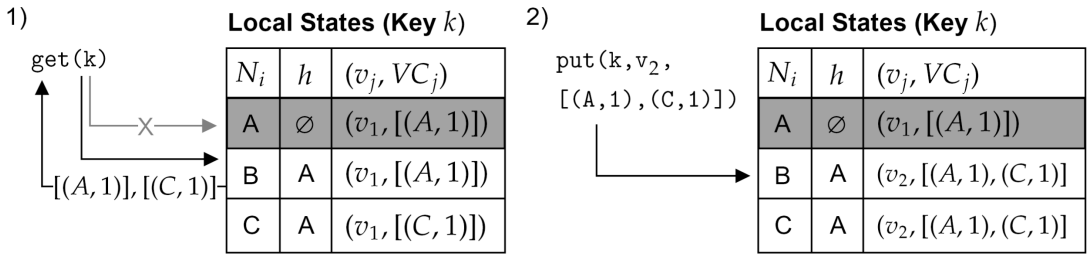
N_i	h	(v_j, VC_j)
A	∅	$(v_1, [(B,1)])$
B	∅	$(v_1, [(B,1)])$
C	∅	$(\emptyset, [])$
D	∅	$(v_1, [(B,1)])$

N_i	h	(v_j, VC_j)
A	B	$(v_2, [(B,1),(C,1)])$
B	∅	$(v_1, [(B,1)])$
C	B	$(v_2, [(B,1),(C,1)])$
D	B	$(v_2, [(B,1),(C,1)])$

N_i	h	(v_j, VC_j)
A	B	$(v_2, [(B,1),(C,1)])$
B	∅	$(v_1, [(B,1)])$
C	B	$(v_2, [(B,1),(C,1)])$
D	B	$(v_2, [(B,1),(C,1)])$

N_i	h	(v_j, VC_j)
A	∅	$(v_2, [(B,1),(C,1)])$
B	∅	$(v_2, [(B,1),(C,1)])$
C	∅	$(v_2, [(B,1),(C,1)])$
D	∅	$(v_2, [(B,1),(C,1)])$

Figure 0.4: 1) A client issues a put request with `(k,v2,[(B,1)])` implying a previous interaction of to B. Though, B is down, and C is the next alive node; However, C has never seen k before. C takes the user's dominating input. 2) C stores the tuple, $(v_2, [(C,1),(B,1)])$, indicating the new value, and notes its participation in a vector clock VC. It propagates (k, v_2, VC) with hint B to D and A, returning a success. 3) Later, B rejoins during gossip. 4) A participants notice B's revival, sending it updates.



N_i	h	(v_j, VC_j)
A	∅	$(v_1, [(A,1)])$
B	A	$(v_1, [(A,1)])$
C	A	$(v_1, [(C,1)])$

N_i	h	(v_j, VC_j)
A	∅	$(v_1, [(A,1)])$
B	A	$(v_2, [(A,1),(C,1)]$
C	A	$(v_2, [(A,1),(C,1)]$

Figure 0.5: 1) The client requests key k from A, but it's down, falling over to B, which gets a read from C; However, C's entry is divergent, perhaps due to a network partition. To resolve, B sends back both vectors, relying on the client to reconcile the data. 2) The client mends the data by merging the two vectors, pushing it back to B. For example, a customer may have had two tabs open with different shopping carts. The application decides to merge the two carts, adding to the user experience.

In an eventually-consistent store, vector clocks track the history of individual object updates but cannot tell us which keys out of millions have diverged across replicas. Dynamo employs the following technique to solve this problem efficiently:

Definition 1.5: Anti-Entropy via Merkle Trees

Dynamo uses Merkle trees to detect and repair divergent replicas without exhaustive scans:

- **Tree Construction:** Each node builds a binary Merkle tree over its local key-value store. **Leaves** are the cryptographic hashes of individual key-value pairs (or small fixed-size batches of adjacent keys), computed as

$$H_i = \text{hash}(k_i \| v_i)$$

  .Then each **internal node** hashes the concatenation of its two children, where ($\|$) denotes concatenation, and H is a cryptographic hash function (e.g., SHA-1, MD5). The root hash summarizes the entire tree.

- **Root Comparison:** Replicas exchange only their root hashes. A match means the entire key-value set is identical—no further work.

- **Recursive Descent:** On a root mismatch, only the child hashes of the differing subtrees are exchanged, descending the tree until the exact leaf hashes (and thus specific keys) that differ are isolated.

- **Selective Synchronization:** Once out-of-sync keys are pinpointed, only those individual (k, v) pairs (and their vector-clock metadata) are transferred and reconciled.

- **Efficiency Gains:** By limiting data exchange to mismatched branches, this method drastically reduces both network bandwidth and disk I/O compared to full scans.

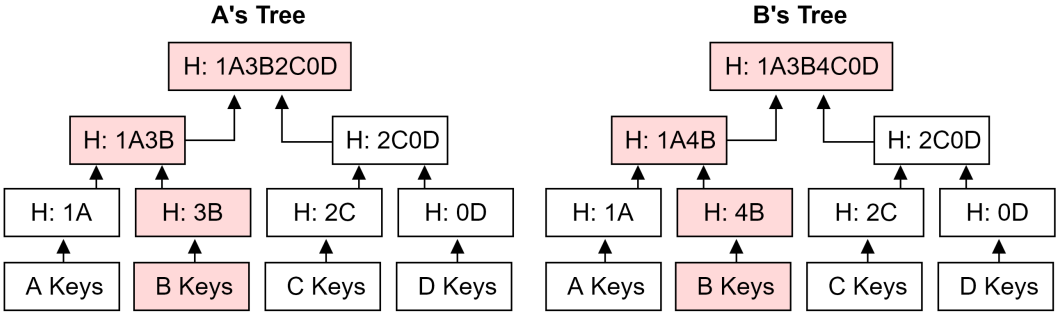


Figure 0.6: Given some arbitrary simplified hashing function, we construct a Merkle tree over key ranges A–D. Here A and B compare root hashes. They differ, so they descend to the branch which causes the mismatch. They both discover a discrepancy in their B key ranges.

Bibliography

[1] Ioannis Liagouris. Cs351: Distributed systems. Lecture notes, Boston University, Spring Semester, 2025. Boston University, CS Department.