

**Updated on 10/21:**

1. Added clarifications to the **Is** command.
2. **Failure Model - Persistence**: Added a footnote on using existing libraries to implement persistence.

CS425, Distributed Systems: Fall 2026  
Machine Programming 3 – Distributed Key-Value Store

Release Date: Oct 13, 2026

Due Date (Hard Deadline): Sunday, Nov 8, 2026 (Code+Report due at 11.59 PM)

*Demos on Monday, Nov 9, 2026*

**This is an intense and time-consuming MP! So please start early! Start now!**

FakeCompany Inc. (MP2) just got acquired in a hostile takeover by the fictitious company ExSpace Inc. Anyway, ExSpace loved your previous work from MP2, so they've re-commissioned you to build a Distributed Key-Value Store (DKVS) for them. DKVS will run on 10 machines in ExSpace's spaceship as it orbits the Earth. This time, ExSpace wants to choose between strong and eventual consistency and understand the performance and correctness tradeoffs. DKVS is based on Cassandra; you can look at Cassandra docs and code, but you cannot reuse code directly from there (we will check using Moss).

**LLM Usage.** Only for this MP, you are allowed to use LLMs for any part of your MP **except actual coding**. In other words, you are free to use LLMs for any, all, or none of designing, writing test code, testing, debugging, and running experiments in your MP. But you're responsible for writing the actual code. If you choose to use LLMs, here are some recommended ways to use LLMs. (1) Design: Come up with a rough draft of the design, and then use your favourite LLM as a sounding board to iterate on and improve that design, and to break it down into modules. (2) Testing: You may then use LLMs to generate unit test cases for each module, and then eventually integration tests. (3) Debugging: You may use LLMs to run the tests against your code and help debug your code. (4) Experiments: you can use the LLM to run your experiments, collect data, and generate the graphs, but the rest of the Report, including the analysis of the graphs/data must be your own.

Crucially, you must document your use and experience with the LLM in the Report. Additionally, you must submit logs that contain your prompts to the LLM and the LLM responses (please see instructions in the report section).

Given the increased use of LLMs to generate code in the industry, these rules for MP3 serve an educational purpose. However, note that, as in your future job, you and only you are responsible for the correctness and efficiency of any design & code that you submit. Additionally (as in your future job), you must understand and be able to explain your code to (your tech-lead or manager) the TA during the Q&A portion of your demo slot. If something fails in the demo, it is your responsibility, not the LLM's. If you're unable to explain your code, you lose points. Proceed cautiously and judiciously.

You must work in groups of two for this MP. Please stick with the groups you formed for MP2. (See end of document for expectations from group members.) Remember – Move Fast and Break Things! (i.e., build some pieces of code that run and incrementally grow them. Don't write big pieces of code and then compile them all and run them all!)

This MP requires you to use code from both MP1 and MP2.

**System Overview:** DKVS is intended to scale with the number of machines. DKVS uses **consistent hashing** to map machines (aka nodes) and keys to points on a ring. DKVS uses a pre-determined hashing function to map nodes and keys to the ring. A key is replicated on the first  $N$  successor nodes in the ring (just like in Cassandra), where  $N$  is the replication factor. Each node uses a membership protocol to maintain/update the full membership list (use distributed group membership from MP2). Thus, given a key, a node can, in  $O(1)$  time, route the request to one of the replicas that contains the key. Note that you should **not** store the list of all key-node mappings at each node. Just the membership list of nodes and their IDs is sufficient to route requests correctly.

Implement single-key put and get operations.

- PUT(key, value): If the key already exists in the store, replace the value. If not, insert this new pair.
- GET(key): If the key exists in the store, return the value. Else, return NOT\_FOUND.
- Values are opaque bytes; support values up to 1 MiB. Empty bytes are a valid value and must be distinguishable from NOT\_FOUND.

All commands should be executable at any of the VMs, i.e., any VM should be able to act as both client and server. Make your design as performant as possible. Allow for maximum concurrency of reads and writes. You cannot take global locks. Note that get and put requests from multiple client VMs may be in progress on the same key.

You must design and implement three configurations:

**Strong:** A linearizable key-value store

**Eventual-noRR:** Eventual consistency with no read repair.

**Eventual-RR:** Eventual consistency with read repair.

**Failure Model.** Unlike MP2, key-value stores typically assume a fail-recover model (as in Cassandra, see lecture notes). Make all data (including version numbers and timestamps) durable to disk before acknowledging so that they survive process crashes<sup>1</sup>. And recover them before serving requests. In other words, you should not delete data on disk when a process simply crashes and restarts. Similarly, a member must not be deleted from the membership list on failure; it is just marked *down* on failure detection. Upon subsequently learning that the node is alive (via heartbeat/gossip), the member is

---

<sup>1</sup> We do not expect you to implement an LSM tree, SSTables, or a full local storage engine. The requirement is simply to persist values and their version metadata before acknowledging writes, so they survive process crashes and restarts. **You are allowed to use an existing library for local storage persistence as long as you clearly document it.** Note you may not use a library that implements distributed replication, quorum operations, or read repair.

marked *up*. The simplest way to deal with this for your MP2 without much change is to keep your suspicion timeout infinite and let the node be marked suspected on `t_fail` and have a node send refute messages upon restart after a crash. A node is deleted from the membership list only if it explicitly leaves.

In all three modes, however, your MP doesn't need to deal with nodes leaving (and fully exiting the membership) and joining. That is, after the initial cluster setup, the ring and each key's replicas remain fixed throughout the run. You only handle nodes crashing and restarting. Failed nodes retain their ring positions and are not replaced by other nodes. Similarly, restarted nodes retain their ring IDs.

Like previous MPs, when we ask you to “kill” (or fail) a node, please force-kill or crash the process(es) (using `kill -9`). We will also ask you to restart a failed process, upon which you will restart it on the same VM with its local disk contents preserved.

### **Configurations.**

Use replication factor  $N = 5$  in all three modes.

**Strong consistency.** Strong mode provides **linearizability** for each key, even under concurrent clients, message delays, network partitions, and crashes/restarts. Each completed operation must appear to take effect at one point between its invocation and response. In linearizability, all operations on a given key are totally ordered in a way that respects real-time order of operations: if A completes before B begins, A is ordered before B. Thus, a get returns the value from the latest put preceding that get in that ordering, or NOT\_FOUND if none exists. Note that a simple quorum intersection discussed in the lecture in the context of Cassandra alone will not yield linearizability. To guarantee linearizability, you have to implement a multi-writer atomic-register quorum protocol, such as the two-phase Attiya, Bar-Noy, and Dolev (ABD) algorithm. The algorithm still relies on quorum intersection, but reads and writes involve two phases. You can find a simple pseudocode here:

<https://muratbuffalo.blogspot.com/2012/05/replicatedfault-tolerant-atomic-storage.html>

To test if your implementation provides linearizability, you can use some openly available linearizability checkers and testing frameworks like the following ones. Porcupine (<https://github.com/anishathalye/porcupine>) and Jepsen (<https://github.com/jepsen-io/jepsen>). We don't mandate this but highly recommend it to catch bugs. Note that you may use LLMs for this testing part.

**Eventual mode (Eventual-noRR):** Use configurable  $R$  and  $W$  with default  $R = 2$  and  $W = 2$ ,  $N = 5$ . Send a put to all reachable replicas, acknowledge after  $W$  distinct qualifying durable acknowledgments, and continue asynchronous propagation after acknowledgment. A get returns the most recent version of the key among its  $R$  distinct qualifying responses; if  $R$  cannot be reached, it does not report success. As in Cassandra, use timestamps for versioning and use the latest timestamp wins policy for resolving conflicts when there is more than one version for a key. Ensure timestamps are unique by using (timestamp, processID) pairs as the actual timestamps.

**Read repair:** When enabled in Eventual mode, the system runs in **Eventual-RR** mode. The response to the client is sent once the get operation has received R qualifying responses, but then the get must compare versions from all reachable replicas. This background comparison must discover the maximum version across these copies and repair stale or missing copies. You must log the returned version, all reachable replicas, received responses from the reachable replicas, and any subsequent repair.

Traditionally, eventual consistency requires replicas to converge once writes stop and failures are fixed. However, for this MP, we relax this guarantee after failures: background synchronization and hinted handoff are outside the scope. Beyond asynchronous write propagation to all reachable replicas, the only repair mechanism required is read repair in Eventual-RR. In Eventual-noRR, replicas that miss writes while down may remain stale after restarting.

When switching between consistency modes, retain membership and ring IDs. However, switching between Strong and Eventual modes clears all KV data and version metadata on every node. When switching between Eventual-RR and Eventual-noRR, you may retain or clear data. Note that ordinary node crashes and restarts must still preserve stored data and version metadata

Think through all failure scenarios carefully and ensure your system does not hang. If an operation cannot obtain the required responses within a timeout of your choice, return an explicit timeout/error status, not NOT\_FOUND.

All commands should be executable on any of the VMs, i.e., any VM can act as both client and server.

For demo purposes, you must implement the following operations:

1) `put key localfilename`: Create or update key with bytes (all the content) from `localfilename`, which refers to the file on the local file system at the client. Print status, ID of the key in the ring, and the acknowledging nodes. For testing and the report, use your own input files of varying sizes. We will provide the input files to use in the demo as part of the demo instructions.

2) `get key localfilename`: Fetch the value of the key and write it to a local file. Print status and returned version to the terminal. If the key is not in DKVS, return NOT\_FOUND status.

3) `ls key`: List all machine (VM) addresses (along with the VMs' IDs on the ring) where this key ~~should be is currently~~ stored. Also print the ID of the key on the ring. **Computing this from the membership list is sufficient; you do not need to query the replicas to check their availability or whether they actually store the key.**

4) `liststore` (at any process/VM): List the set of key names (along with their keyIDs) that are replicated (stored) on DKVS at this process/VM. Also, print the process/VM's ID on the ring.

5) `getfromreplica VMaddress key localfilename`: Fetch only that replica's local value for the key, and store it locally on `localfilename`, but do not perform any further actions, such as read repair. Print status and returned version to the terminal. We will use this on all replicas to check if values and versions are identical across replicas in the eventual mode (e.g., after read repair). If the key does not exist on the replica, return `NOT_FOUND` status.

6) `list_mem_ids`: Augment `list_mem` from MP2 to also print the ID on the ring that each node in the membership list maps to. You must sort this by `nodeID` (useful for tests).

7) `multiput(key, VMi,...,VMj)`: Launches puts from `VMi,...VMj` simultaneously to the same key; each VM can choose an arbitrary local file with random bytes as the value for its put.

8) `consistency_mode {strong, eventual-noRR, eventual-RR}` Switches between the aforementioned 3 modes. This will change the mode in all processes.

For each of the commands, please print terminal-level messages at the initiating client that show when the command has been completed. This should be shown when the operation completes in DKVS. For a put, also list the VMs to which the key was replicated. Each replica that receives a get or put request must print messages to its terminal: when it received the request and when it responded to the request. The printed message format must be `<local timestamp>: <received | responded><get | put> <key>`.

Create logs on each machine. You can make your logs as verbose as you want, but at the very least, you must log each time a key operation is processed (once when a node receives a request and once when the node completes the operation). We will ask to see the log entries during the demo via the MP1 query mechanism. You should also use your MP1 solution for debugging MP3.

Use your MP2 code to maintain membership lists across machines and detect failures. During the demo, we may need to invoke some MP2 commands, e.g., to list membership.

We also recommend (but don't require) writing unit tests for the basic key operations. Ensure that your system works for a long series of key operations.

**Language:** Choose your favorite language! We recommend one of C++/Java/C/Go/Rust (but you are free to go beyond this list!). We will release "Best MPs" from the class in these languages only (so you can use them in subsequent MPs).

**Report:** Write a report (typed, not handwritten) of less than three pages (12 pt font). Briefly describe the following (we recommend your report contain headings with bold keywords):

a) **Design:** (no more than a page) focus especially on the algorithm and design used, and how you met the requirements – consistency guarantees, versioning, recovery, and read repair;

b) **LLM use:** If you used LLMs, describe in <300 words your usage and experience---especially how beneficial or not it was. Full LLM transcripts must be submitted with the code. Place them in a folder named `llm_use/` in your repo. Instead of manually copying prompts/responses, you can use log-extraction tools. [We provide one here.](#)

c) **Measurements** (measure real numbers; do not calculate by hand or calculator!): see questions below. For each plot/data, please use the terms in brackets so your report is easy to read.

- (i) **(Performance):** Compare Strong and Eventual without read-repair. Configure Eventual-noRR in two ways:  $R = 1, W = 1$  and  $R=2, W=2$ . Use all 10 VMs and  $N = 5$ . Use values of size 1 KiB with 1, 2, 4, 8, and 10 concurrent clients in a healthy cluster. Plot throughput on the y-axis against mean latency on the x-axis, with one curve per configuration and one labeled point per client count. Preload 100,000 keys and run a workload with a mix of 50% get and 50% put operations that choose keys randomly from the key range with uniform probability, and with unique values. **Each client should run the workload for at least 3 minutes.**
- (ii) **(Performance with read repair):** Compare Eventual with and without read repair for  $R = 1, W = 1$ . Use all 10 VMs and  $N = 5$ . Use values of size 1 KiB and 10 concurrent clients. Use separate bar plots to compare throughput and mean latency. Preload 100,000 keys and run a workload with a mix of 50% get and 50% put operations, but with 10 keys being very popular. Specifically, 99% of get and put operations pick one key randomly from only those 10 keys. The remaining 1% of operations pick randomly from the rest of the keys. Each client should run the workload for at least 3 minutes.

Both experiments use a healthy cluster (no failures). Each client issues one operation at a time. Report aggregate successful operations per second and mean end-to-end latency across all clients. Complete preloading on all replicas before measurement begins. Reuse the same generated workload seeds across the compared variants.

For each data point, take at least three readings, and plot averages **and** standard deviations (and, if you can, confidence intervals). Discuss your plots, don't just put them on paper, i.e., discuss trends briefly and whether they are what you expect or not (why or why not). Measurement numbers don't lie, but we need to make sense of them! Stay within the page limit!

**Submission:** Similar to past MPs. More instructions on Piazza, but the checklist/workflow is the same. Do not miss the deadlines for signing up for the demos (usually the Friday before submission Sunday), or to submit the form and the Gradescope report (both due on Sunday).

**When should I start?** Start NOW. You already know all the necessary class material to

do this MP. Each MP involves a significant amount of planning, design, and implementation/debugging/experimentation work. **Do not** leave all the work for the days before the deadline – **there will be no extensions**.

**Evaluation Break-up:** Demo [60%], Report (including design and plots) [30%], Code readability and comments [10%].

**Submission:**

(1) **Report.** Submit your **report** to Gradescope BEFORE the deadline, and **TAG** your page(s) on Gradescope (there will be a penalty if you don't). Only one group member should submit – you must tag your partner.

(2) **Demo.** There will also be a **demo** of each group's MP3. Sign-up sheets for the demo & demo instructions will be posted on Piazza. Signup for demos on the Friday before the deadline.

(3) **Code.**

A. Submit your working **code** into a subfolder named mp3 in your **GitHub** course repo. In your repo, please include a README explaining how to compile, how to run your unit/demo tests, and how to run your MP3 code along with LLM transcripts.

B. Fill out the MP3 **code** submission **Google form**.

Further submission instructions will be posted on Piazza.

**When should I start?** We strongly recommend that you start early. Each MP involves a significant amount of design, implementation, debugging, and experimentation work. **Do not** leave all the work until the end.

**Academic Integrity:** You cannot look at others' solutions, whether from this year or past years. We will run Moss to check for copying within and outside this class – first offense results in a zero grade on the MP, and second offense results in an F in the course. There are past examples of students penalized in both those ways, so just don't cheat. You can discuss only the MP spec and lecture concepts with other students and on the forum, but not solutions, ideas, or code. If we see you posting code on the forum, that's a zero on the MP.

Contributions by Group Members: We recommend you stick with the same group from one MP to the next (this helps keep the VM mapping sane on IT's end), except for exceptional circumstances. We expect all group members to contribute equally to the overall effort. If you believe your group members are not, please have "the talk" with them first, and give them a second chance. If that doesn't work, please email both instructors.

**Happy Key-Value Storing (from us and the fictitious ExSpace)!**