

P3: Replication for Fault-tolerance

✓ Published

👤 Assign To

✎ Edit

⋮

Overview

This project is the third (and the final mandatory step) in a series of projects where we practice distributed system coding and build **MadKV** (<https://github.com/josehu07/madkv>), a distributed, partitioned, replicated, fault-tolerant, and performant key-value store in steps.

In this step, we will build upon our Project 2 solution and add one more critical feature: *replication* for fault tolerance and availability. So far, our KV store service offers no *redundancy*. Every piece of information is stored only once in the system. If a partition server goes down, requests involving that partition cannot make progress until a new server takes over, which takes a long time. If the durable storage of that partition becomes inaccessible, recovery can be tricky. If the manager goes down, new clients cannot connect to the service at all. In short, our KV system right now offers quite low availability.

To improve, we adopt the idea of *state machine replication* and make each existing component of the KV service a replicated "cluster". Failing one node is common; failing two or more nodes at the same time in a replicated group is exponentially rarer. Say we choose a *replication factor* of r . Each partition is now composed of a group of r server replica nodes, appearing as one fault-tolerant "server" that almost never fails. Making each partition replicated is the baseline requirement; for bonus points, the manager would also be composed of a group of manager replicas.

You will use your Project 2 solution as the starting point. In case we have made updates to the project template, pull our updates to the **main** branch from the GitHub repo: <https://github.com/josehu07/madkv> (<https://github.com/josehu07/madkv>), and merge into your development branch. See the README for instructions.

As usual, all actions relevant to grading should be made invocable through **Just** (<https://github.com/casey/just>) recipes. You will need to fill out the blanks in the project-level **Justfile** to surface your own code.

Project Groups Policy



You can work on this project with exactly one other person. You can work with the same person as you did for Project 2 or someone new. Just be sure you communicate these decisions clearly with your previous project partner!

Once you've finalized your project partner, place yourselves in the same "Project 3" Group (which we have defined) in Canvas.



You are welcome to talk with others in the course about the project. You may use piazza to ask any level of question.

Consensus Protocol

Quorum-based *consensus* protocols, such as *MultiPaxos* or *Raft*, are the de facto standard of linearizable replication in today's consistency-critical distributed systems. They use the mathematical properties of quorum intersections to help establish agreement across nodes. In this project, we follow standard practice and deploy the consensus protocol in the form of *state machine replication*: the replicas agree on an ordered *log* of incoming commands. This log is the essential data that's being replicated and made durable, not the actual state data. Replicas apply committed log entries to their (possibly volatile) state at their own pace.

When implemented correctly, each replicated group satisfies **linearizability** ([ref1](https://dl.acm.org/doi/pdf/10.1145/78969.78972) , [ref2](https://arxiv.org/pdf/2409.01576) , <https://arxiv.org/pdf/2409.01576>.) so the group appears as a single entity to other components of the system. At the end of this project, your KV system should look exactly like what you had at the end of Project 2, just with each component supported by r collaborative replicas and failing much less often. All the consistency semantics discussed in Project 2 still hold. Assume we always choose an r among 1, 3, 5, 7, or 9.

Option 1: MultiPaxos


MultiPaxos , <https://lamport.azurewebsites.net/pubs/paxos-simple.pdf> is a classic consensus protocol. At its core is **Paxos** , <https://lamport.azurewebsites.net/pubs/lamport-paxos.pdf>, a single-decree consensus protocol enabling agreement on a single value across acceptor nodes. MultiPaxos incorporates Paxos and extends it to support efficient replication of an ordered log. We have covered Paxos in this course, so we will not include much technical detail in this project spec. Please refer to the paper and other course materials, and contact us if you have any questions about the theory behind the protocol.

If you choose to implement MultiPaxos, make the following assumptions:

- Assume replicas are symmetric (although Paxos can be deployed in a more extensible way). Each replica plays the role of an *acceptor* and a *learner*. Additionally, at any time, one of the replicas (the one that proposed the highest ballot number) acts as the *distinguished proposer*; this replica is the *leader* that serves client requests.
- Assume the Prepare phase covers all log entries up to infinity as described by *batched* Prepare in the paper. In the normal case where a leader and a majority quorum of replicas are healthy and connected, new leader step-ups should be avoided, and only the *Accept* phase should happen on the critical path of a client request.

Heartbeat messages will be necessary for the detection of unresponsive replicas. We are happy to answer any questions related to the design of the MultiPaxos protocol.

Option 2: Raft

Raft , <https://raft.github.io/raft.pdf> is a popular consensus protocol among newer distributed system implementations. It shares the same core with Paxos-style protocols, but introduces two explicit design features: 1) strong leadership based on term number + log indices and 2) randomized election timeouts. The paper does a great job of presenting the protocol in one condensed document and improving *understandability* for readers.

For our project, the RPC descriptions presented in the paper should be more than enough for you to implement a minimal working prototype of linearizable replication. If you had little experience writing

distributed system code prior to this course, you may find the Raft paper easier to follow than MultiPaxos materials scattered around the web. However, if you find the strong leadership design (especially the log index part) a bit confusing, be aware that this design is indeed less intuitive and more restrictive than vanilla Paxos. In this case, you may find MultiPaxos actually easier to reason about and understand. Pick the one you find more comfortable implementing.

You should NOT use any high-level libraries that implement a consensus protocol for you; those defeat the purpose of the project. You should NOT copy any code from public codebases (you can find plenty of open-source Paxos and Raft implementations online; you are very welcome to use them as a reference, but you must build your own implementation atop your Project 2 result). When in doubt, please don't hesitate to ask.

Bonus: Replicated Manager [+15%]

As a bonus, make your manager replicated as well, with its own configurable replication factor. This would require you to have a "generic" SMR layer implementation shared across your manager executable and server executables. More details in the "Code Structure Suggestions" section below.

Bonus: Chain Replication for Servers (in addition to manager replication) [+15%]


Chain Replication

(https://www.usenix.org/legacy/publications/library/proceedings/osdi04/tech/full_papers/renesse/renesse.pdf) is a classic approach to achieving high-throughput linearizable replication. It organizes replicas in a *directed chain* topology. All writes arrive at the head node, get ordered by the head node, and propagate down the chain; writes are acknowledged only after they've passed the tail node. All reads are served by the tail node. We won't include much technical details about the protocol in this project spec. Please refer to the link paper and relevant course material.

Chain Replication is relatively easy to reason about and implement, and can offer excellent throughput with pipelining and higher tolerance to server failures. However, it has a few insufficiencies by itself:

- Writes experience long latency as they have to pass through all r replicas before acknowledgment.
- Reads have to contend on the tail node (although many consensus protocols exhibit the same drawback, there exist many techniques to mitigate).
- It never hides failures transparently: if a node on the chain fails, new writes will be blocked by it. The protocol must be able to detect such suspicious nodes and remove them from the chain promptly, which takes time and requires a standalone *master oracle*.
- Most importantly, making the master oracle itself fault-tolerant requires -- you guessed it -- a consensus protocol.

We offer the option to implement Chain Replication for the KV servers, using the KV manager as the oracle for the chain. This means you need to implement two protocols: a consensus protocol of your choice for the manager/master, plus chain replication for the servers of each partition. This is a challenging task to complete in the tight timeframe of a course project; prioritize the consensus protocol implementation.

Both MultiPaxos and Raft are leader-oriented protocols. If you have an advanced protocol in mind that you want to implement, e.g., a leaderless protocol such as **EPaxos** ,

(<https://www.cs.cmu.edu/~dga/papers/epaxos-sosp2013.pdf>) or other modern variants of Paxos or Raft, and you think the protocol satisfies our requirements, let us know and we will be happy to discuss.

Availability Behavior

Similar to the previous project, we assume a fixed, static cluster topology at launch time. This means the following configurations are known at launch time and are kept static:

- The number of keyspace partitions
- The replication factor of the manager (if supported)
- The replication factor of data servers
- The address of every node in the KV service


Your replication protocol implementation only needs to work with the above assumption. Membership changes (i.e., reconfigurations) are a hard topic and we don't require solving them in mandatory projects.

Failures & Expected Guarantees

Defining the *failure model* is a crucial part of distributed systems study. In this project, we only require dealing with node crashes and unresponsiveness. A replica node could crash and lose connection to other nodes. A node could also become super slow and/or completely unresponsive to others. Peer nodes should deploy appropriate *timeouts* to detect and mitigate failed nodes. You can use generous timeout lengths in the magnitude of seconds.

Your replication implementation should behave gracefully in the presence of node failures:

- Within a replicated group, any *minority* number of failed nodes should not block the availability of the group indefinitely. Given an odd replication factor r , the group should continue to provide service if $f \leq \lfloor \frac{r}{2} \rfloor$ nodes fail. There might be a brief downtime if a leader failed and a new one had to step up, but the system must keep making progress after that.
- Under any higher number of node failures in a group, the group as a whole could hang and become unavailable, but it should never return inconsistent results for client requests.

In real systems, the problem would be much more complex when asynchronous network issues are involved (messages could arrive out-of-order and could be dropped). Here, it is safe to think that the network is "fairly reliable": just use TCP connections and assume all TCP properties hold. We also exclude any [Byzantine faults](https://en.wikipedia.org/wiki/Byzantine_fault)  (https://en.wikipedia.org/wiki/Byzantine_fault) and assume that all nodes are collaborative. There will be no malicious nodes and the network never tempts with messages.

You don't have to implement replica node recovery and re-joining to resume the cluster size; bonus points are available if you do implement it. This feature won't be too hard to add given a decent implementation of the rest of the replication protocol. Nonetheless, we don't enforce correct node re-joining.

A Note on the Client Side

Both MultiPaxos and Raft are leader-based protocols. Raft has explicit leadership which at most one node will hold at the same time. MultiPaxos has implicit leadership: there could temporarily be multiple nodes proposing

higher ballot numbers and stepping up as leader, but they should quickly resolve to obey the highest-ballot proposer in the absence of failures.

In either case, clients of a replicated group of servers are given the list of addresses of all replicas. To find which node is the effective leader to send requests to, some logic needs to be implemented on the client side (note that the "clients" -> "servers" relationship here could mean the KV clients -> KV managers for inquiries, KV clients -> KV servers of a partition, or KV servers -> KV manager for registration).

A common implementation is to let a client pick any server to start with. If the server does not think of itself as leader, it replies to the client with a redirection message to tell the client who is the current effective leader of the cluster; if none exists, it tells the client to try on any other server. Once a leader is found, the client sticks to that node for requests. If the leader node ever steps down or fails at some point, the client should be able to detect that (through broken connections and/or timeouts), and choose another server to repeat the above procedure until a new leader is established and found.

Code Structure Suggestions

Clear abstractions and modularization of code are good practices in system programming. This project gives you an opportunity to practice this yourself. All the KV partitions (and the manager, if supported) need to be replicated, so they share the same demand for a state machine replication abstraction. We suggest you implement your replication protocol infrastructure as a library (or other form of shared code) that is generic over the state machine commands. Different components of your KV service then simply instantiate new replica groups and drive them with their commands.

Command Line Arguments

A manager replica, server replica, and client executable should expect the following collection of arguments (argument names and values are just examples):

- Manager: `./yourmanager --replica_id 0 --man_listen 0.0.0.0:3666 --p2p_listen 0.0.0.0:3606 --peer_addrs 8.7.6.5:3607,12.11.10.9:3608 --server_rf 3 --server_addrs 1.2.3.4:3777,5.6.7.8:3778,9.10.11.12:3779,13.14.15.16:3780,17.18.19.20:3781,21.22.23.24:3782 --backer_path ./backer.m.0`
 - `replica_id` is the index of this replica in the replicated group of managers (if you do not support manager replication, you will only have replica 0 here)
 - `man_listen` is the address to listen on for client and server inquiries and registrations
 - `p2p_listen` is the address to listen on for peer replica connections for protocol-internal use
 - `peer_addrs` is a comma-separated list of peer replicas addresses to connect to, sorted by replica ID, excluding self. Since this list could be empty when the replication factor is 1, a special string "`none`" is used to indicate an empty list
 - `server_rf` is the replication factor of a partition (this may be different from that of the managers; the managers' rep factor is inferred as the length of the `peer_addrs` list + 1)
 - `server_addrs` is a comma-separated list of all server nodes, indexed first by partition ID then by local replica ID
 - `backer_path` is the path to my durable storage directory

- The example command given above indicates that we are starting a manager replica who is the replica 0. It listens on `0.0.0.0:3666` for server registrations and client inquiries. It listens on `0.0.0.0:3606` for peer manager replicas to connect. Replica 1 is at `8.7.6.5:3607` and replica 2 is at `12.11.10.9:3608`. There are two partitions ($6 / 3 = 2$), each replicated three ways. The first partition's replicas API addresses are `1.2.3.4:3777`, `5.6.7.8:3778`, and `9.10.11.12:3779`. The second partition's replicas API addresses are `13.14.15.16:3780`, `17.18.19.20:3781`, and `21.22.23.24:3782`. Durable storage is under directory `./backer.m.0/`.
- Server: `./yourserver --partition_id 0 --replica_id 0 --manager_addrs 1.2.3.4:3666,8.7.6.5:3667,12.11.10.9:3668 --api_listen 0.0.0.0:3777 --p2p_listen 0.0.0.0:3707 --peer_addrs 5.6.7.8:3708,9.10.11.12:3709 --backer_path ./backer.s0.0`
 - `partition_id` is the index of the partition that this server belongs to
 - `replica_id` is the index of this replica in the replicated group of servers in the partition
 - `manager_addrs` is a comma-separated list of manager replicas management API addresses to connect to for registration
 - `api_listen` is the address to listen on for client KV operations
 - `p2p_listen` is the address to listen on for peer replica connections for protocol-internal use
 - `peer_addrs` is a comma-separated list of peer replicas addresses to connect to in my partition, sorted by replica ID, excluding self. Since this list could be empty when the replication factor is 1, a special string "`none`" is used to indicate an empty list
 - `backer_path` is the path to my durable storage directory
 - The example command given above indicates that we are starting a server replica who is the replica 0 in partition 0. It listens on `0.0.0.0:3777` for client requests and `0.0.0.0:3707` for peer server replicas. The managers can be found at `1.2.3.4:3666`, `8.7.6.5:3667`, and `12.11.10.9:3668`. Server replica 1 in my partition is at `5.6.7.8:3708`, and replica 2 is at `9.10.11.12:3709`. Durable storage is under directory `./backer.s0.0/`.
- Client: `./yourclient --manager_addrs 1.2.3.4:3666,8.7.6.5:3667,12.11.10.9:3668`
 - `manager_addrs` is a comma-separated list of manager replicas management API addresses to connect to for server information inquiries

As usual, the argument names are not important; fill in the just recipes to connect our recipe parameters to your executable invocation commands.

Correctness Testing

You will craft a few interactive testcases (see below for descriptions) and also run our fuzz tester with some required configurations.

Your Own Testcases

The implementation work for this project is rather heavy, so we will only ask you to prepare four testcase scenarios that you will run during demo time:

- Launch a service cluster with a proper manager replication factor, a proper number of partitions, and a proper server replication factor. Show that your client can make and complete requests. Then, fail some

($\leq f$) *follower* servers in at least two partitions, and show that your client can still make and complete requests to those keys and get consistent results.

- Launch a service cluster similarly. Repeat the same experiment but fail the *leader* node at least once. Election should be triggered transparently after a leader failure, and clients should be able to continue operations after the election completes.
 - To find which server is the leader, you may, for example, simply let your server executable print leader election results to stderr.
- (If manager replicated,) Launch a service cluster similarly. Show that failing a manager node does not prevent new clients from joining and using the KV service.
- Launch a service cluster similarly. Pick a partition and fail $> f$ servers (i.e., more than the availability threshold of your protocol) in the partition. Show that client requests to that partition now hangs (or times out), instead of returning inconsistent results.

You don't need to prepare automated recipes for these testcases. You just need to be able to demonstrate them during the in-person demo.

Fuzz Testing

We will reuse the fuzz tester utility for fuzz testing. This time, we always run 5 clients with conflicting keys. We ask you to manually set up the following two scenarios on the service side:

- Launch a service with a server replication factor of 5 (and pick values you like for other parameters; one partition is fine), then run the fuzzer till completion without abnormality.
- Similarly, launch a service with a server replication factor of 5, and run the fuzzer. This time, pick a partition and gradually crash two servers in that partition (one of them must be the leader at that time). Fuzzing should still complete successfully.

Make sure to empty the durable storage directories before starting the service, so that it starts from a fresh, empty state.

CloudLab has limited resources, so you will find it hard to start experiments with a lot of machines. It is totally fine to run testing & benchmarking on a smaller cluster and have some (or all) servers/replicas share the same machine. In the report, write clearly the physical topology you used in testing and benchmarking.

Performance Benchmarking

Similar to project 1 & 2, we will run [YCSB](https://github.com/brianfrankcooper/YCSB)  (<https://github.com/brianfrankcooper/YCSB>) benchmarks through our benchmarker utility and report some performance trends. Focus on relative trends across your own results; the absolute numbers may vary due to various factors (physical instance type, implementation language, etc.) and are not important. We recommend CloudLab instance types with ≥ 20 CPU cores, ≥ 128 GB memory, and a decent disk, running Ubuntu 22.04.

As part of the deliverables, you will report the following results:

- 10 clients on all workloads A to F, varying the server replication factor across 1, 3, and 5 (pick values you like for other parameters; one partition is fine)
- workload A with 1 or 5 server replication factor, scaling the number of clients from 1 to 30

Our report generation script will list all configurations needed for plotting. Make sure you understand the benchmarking results and comment with your observations.

Deliverables & Grading

You will deliver both your code and a short report at the end of this project. We will grade the project by having each group do a short (10 min) demo with us, where you walk through your code and report, and check the grading bullet points with us one by one.

Archived Codebase

Bundle your codebase into a single 'tar.gz' archive to submit it through Canvas. When archiving, run the following command at the root level of your repo (i.e., where you find `README.md` and the top-level `Justfile`).

```
tar --exclude-vcs -zcvf <somename>.tar.gz ./*
```

Verify your archive by untaring it with:

```
mkdir -p ../testrepo
mv <somename>.tar.gz ../testrepo/
cd ../testrepo
tar -zxvf <somename>.tar.gz
ls # should see README.md
```

Project Report

The code will be graded together with a short report PDF, which should contain the following information. Please be concise when explaining your design or commenting on your testing and benchmarking results. Restrict each point to a few sentences max.

- Design walkthrough [30%]
 - Code structure & abstractions [10%]
 - What you find easy, hard, and interesting in replication protocol implementation [10%]
 - Specifically, comment on how node failures are mitigated [10%]
- Your own testcases [30%]
 - Failing some server follower replicas [10%]
 - Failing a server leader replica to trigger elections [10%]
 - Failing beyond the availability threshold [10%]
 - Failing a manager replica [if supported, to showcase manager fault-tolerance]
- Fuzz testing [20%]
 - Replicated servers, no crashing [10%]
 - Replicated servers, with crashing [10%]
- YCSB benchmarking & explanation [20%]
 - 10 clients across workloads A to F with 1, 3, 5 server replication factor [10%]
 - Workload A with 1 or 5 server replication factor, scaling #clients [10%]
- Any bonus content [+50% max]
 - Recovery & re-joining of failed replicas to maintain cluster size [+10%]
 - Replicated cluster manager [+15%]

- In addition, Chain Replication for the replication of KV servers [+15%]
- Other advanced engineering of replication system features: *snapshotting*, *membership management*, *read leases*, etc. (don't get overwhelmed by these fancy terms; each of these deserves a full project, if you choose to do it for your open-ended Project 4)
- Additional scripts on top of `just p3::service` and other recipes to help you launch service and do benchmarking
- Any additional testing & benchmarking, unit tests of your code components, etc.

We provide utilities that help run everything and generate a template Markdown report from your code. Please fill the report with text and comments at proper markers. You may submit the Markdown source file (in this case also submit the plot images), or render it into a PDF (recommended) and submit the PDF instead. Your comments should reflect your code *faithfully* and *comprehensively*.

We will grade the project by having you do a short (10 min) demo with us, where you show us your code running live, walk through your report, present your test & benchmark results, and explain your observations.

Points 100

Submitting a file upload

File Types gz, md, pdf, and png

Due	For	Available from	Until
Apr 13	Everyone	-	-

+ [Rubric](#)