TinyKV: A High Performance Distributed Key-Value Store

Final Report for the Distributed Systems Course 2025/26

Emil Rafael Solberg
emilrafael.solberg@studio.unibo.it

February 9, 2026

TinyKV is a high performance distributed Key-Value store implementation. It is written in modern c++ (C++20) and uses grpc, multithreading and fine-grained locking to achieve up to 5000 Operations Per Second. TinyKV implements features like Tunable Consistency, Consistent Hashing, LWW Conflict resolution and asynchronous Heartbeats.

1 Concept

TinyKV is a distributed Key-Value storage system designed as a high-performance, fault-tolerant backend infrastructure component. It adopts a decentralized, peer-to-peer architecture inspired by Amazon Dynamo, prioritizing Availability and Partition Tolerance (AP) over strict consistency.

1.1 Product Type

The product is delivered as a suite of two primary components:

- **Server Nodes:** A server application that form the cluster. Responsible for handling data replication, heartbeats and data storage.
- **The Command Line Interface** A Command-Line Interface (`tinykv_client`) that allows users to interact with the cluster. It is the main entry point for storing and retrieving data.

1.2 Use Case Description

TinyKV operates as a backend storage system, meaning the typical user is often other programs or sysadmins as compared to say mobile devices or browsers.

- **What it does:** Provides a distributed map structure that allows user to write data with a specified replication factor and read data with a specified consistency quorum.
- **Interaction:** Users interact with the CLI through a terminal that has network access to the cluster.
- **Data Storage:** The system stores timestamped key-value pairs. The data is sharded across the cluster and replicated to multiple nodes to prevent data loss.
- **Roles:** The system distinguishes between three roles:
  - **The Coordinator/Owner:** The node that owns the data according to the Hash Ring. If the initial node is not the owner, it simply forward the request accordingly. The coordinator is responsible for performing cluster health checks and orchestrating replication.
  - **Replica:** A node that receives data from a Coordinator. It simply stores the data as instructed.

1.3 Why Distribution?

The two main benefits of distribution in TinyKV is ensuring Fault Tolerance and High Availability.

- **Fault Tolerance**: TinyKV replicates data based on the user specified replication factor (with a default value of 3). Should any single node crash, the data remains retrievable from the replicas.
- **Partitioning:** Through the use of consistent hashing, the system is able to scale in order to exceed the memory limitations of any single machine.
- **Availability:** Because of the read quorum the system can accept reads even in the case of several nodes being offline. Because of this the system eliminates any single point of failure and can remain available as long as a subset of nodes remain active.

2 Requirements Elicitation and Analysis

2.1 Functional Requirements

The requirements below define the functionality the system should expose to the user via the command line interface.

1. **PUT operation**
   - **Description:** Users should be able to use the Put operation to store string based Key-Value pairs. The user must be able to specify a desired Replication Factor (RF).

- **Acceptance Criterion:** A client issues a `put <key> <value> [RF]` command to any node in the cluster and receives an acknowledgment from the system.

2. **GET operation**
   - **Description:** Users should be able to use the Get operation to retrieve any string based value associated with a key that has been stored in the system. The user should be able to specify the Read Quorum (R) size to control the consistency level of the data read.
   - **Acceptance Criterion:** A client issues a `get <key> [R]` command to any node and receive the correct value associated with the specified key.

2.2 Non-Functional Requirements

1. **Availability (CAP Theorem)**
   - **Description:** The system focuses on Availability, and Network Partitioning of the CAP theorem [1]. TinyKV must continue to accept reads and writes even if a minority of nodes are unreachable. Consistency is relaxed but tunable to prioritize up-time.
   - **Acceptance Criterion:** The system successfully completes a put request even after one or more nodes in the cluster have been terminated, provided the remaining nodes meet the replication factor size.

2. **Tunable Consistency**
   - **Description:** In the event of conflicting data, for example because of concurrent writes, the system guarantees strong consistency through LWW provided the Read Quorum is large enough such that $R + W > N$ [1].
   - **Acceptance Criterion:** A read operation with a sufficient read quorum size should always provide the value with the highest timestamp. E.g Last Writer Wins.

3. **Scalability**
   - **Description:** The system should scale horizontally and remain operational and performant even as more nodes are added to the cluster.
   - **Acceptance Criterion:** After adding more nodes the cluster should remain functional, correct and performant after system reboot.

4. **Performance**
   - **Description:** The system must support high-throughput operations with low latency suitable for a backend datastore.
   - **Acceptance Criterion:** The benchmark suite demonstrates a throughput exceeding 1,000 Operations Per Second (OPS) on standard development hardware.

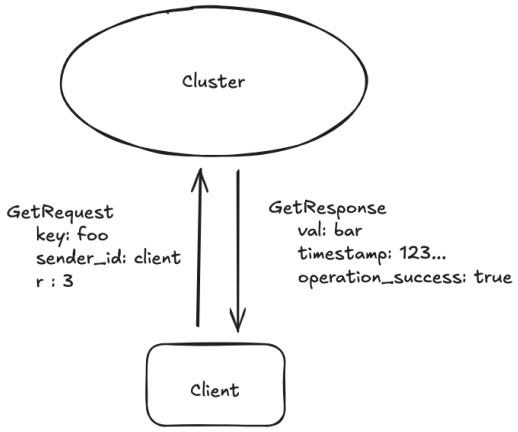

Figure 1: System from the clients perspective

2.3 Implementation Requirements

1. **Programming Language: C++20**
   - **Justification:** C++20 Allows for fine grained control over memory and the host machine, allowing for low-latency execution and high performance. Its also a compiled language removing the overhead of an interpreter compared to a language like Python
   - **Acceptance Criterion:** Source code compiles using a standard C++20 compiler
2. **Communication Protocol: gRPC**
   - **Justification:** Required to ensure efficient communication between entities. Protocol buffers allow for memory efficient serialization and strictly typed contracts for communication.
   - **Acceptance Criterion:** All requests are defined in a .proto file and generated via the `protoc` compiler.
3. **Containerization: Docker**
   - **Justification:** Ensures portability and a stable runtime environment. Standardizes the build system and removes cross platform compatibility issues.
   - **Acceptance Criterion:** The system is managed entirely through docker containers

2.4 Relevant Distributed System Features

- **Transparency**

  - **Location Transparency:** From the clients perspective the cluster is perceived as a single logical entity as seen in Figure 1. No matter which node the client connects to, that node will be able to serve or forward the request as needed. For instance the client may request key "foo" from Node 1, Node 1 consults the Hash Ring and forwards the request to the owner of key "foo" (for example Node 3). The corresponding value "bar" is then returned by Node 1. To The client it appears as if it only interacted with Node 1.
  - **Failure Transparency:** If any node crashes the system will automatically route requests to replicas. From the users point of view the only consequence might be slightly higher latency, but the operation still succeeds.

- **Fault Tolerance & Dependability**
  - **Availability:** The system remains available even if a subset of nodes are unreachable. So long as the majority of nodes remain operational, the system will remain so as well. If any node goes down, the ownership of its data is simply reassigned according to the Hash Ring.
  - **Data Integrity:** To ensure data integrity and prevent stale writes during network partitions, TinyKV employs *Last Write Wins* (LWW). Even if backups receive writes out of order the storage will still maintain the correct state.
  - **Safety:** The replication factor guarantees that data survives even in the case simultaneous loss of nodes. The higher the replication factor the more resistant to loss of data. Although it should be noted that this is a tradeoff between safety and request latency.

- **Scalability**
  - **Horizontal Scaling:** TinyKV is designed to easily scale horizontally by adding more nodes. Through the use of Consistent Hashing and virtual nodes, the key space is distributed equally, allowing the system to support an arbitrary amount of nodes. It should be noted that adding nodes requires a system restart. This is discussed further in 9.

- **Performance & Concurrency**
  - **Communication Efficiency:** The use of gRPC reduces handshake overhead for frequent intra-cluster communication like heartbeats and replication compared to HTTP.
  - **Concurrency:** TinyKV maximizes CPU utilization through multithreading. While one thread is blocked, others can process requests. The benchmark showcases that multi-threading results in upwards of 10x speedup. It should be noted that the performance bottleneck here is not the system itself but rather network latency. By running the application on bare metal as opposed to a Docker container, and removing terminal output, the system should be able to handle requests significantly faster.

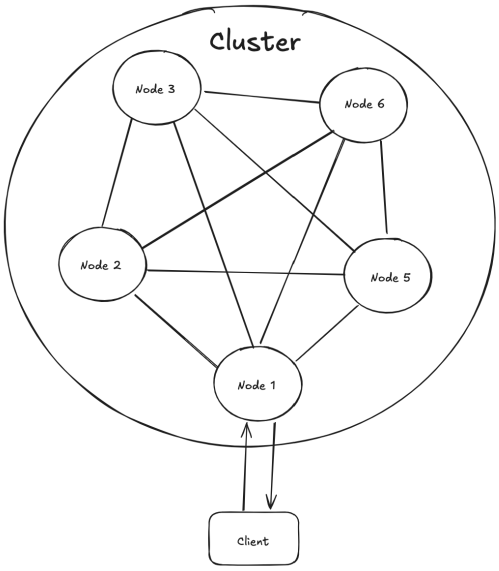

Figure 2: System Architecture. Note that Client can communicate with any node in the cluster

3 Design

3.1 Architecture

- **Architectural Style:** Peer-to-Peer (P2P).
  - **Why:** To achieve the high availability requirement, TinyKV uses a P2P architecture as seen in Figure 2. This eliminates any single point of failure, as all nodes are equal. Any node can handle client requests, orchestrate replication or store data.

3.2 Infrastructure

- **Infrastructural Components:**
  - **Server Nodes** Server Nodes act as the core component of tinyKV. By default 5 instances runs the tinykv-server executable. Each node is equal and can take on the role as both Coordinator or Replica for any given request.
  - **Client:** The CLI tool that connects to any available node to interact with the cluster.
- **Distribution:** Docker simulates a private virtual network that the server nodes are distributed across. A Node runs in an isolated Docker container given a unique address. Docker handles issues like address to ip resolution across the network.

- **Discovery:**
  - **Naming:** Nodes are addressed by their hostname and port (e.g., `tinykv-node1:50051`).
  - **Static Discovery:** On startup the cluster is of a fixed size, determined by the address space (defined in clusters.txt) and the number of instances defined in docker-compose.yml. This provides the initial peer list, used for heartbeats and consistent hashing. The consequence of this is that adding additional nodes requires a system reboot. This is discussed further in section 9.

```
message PingRequest {
  string sender_id = 1;
}

message PingResponse {
  bool is_ready = 1;
}

message PutRequest {
  string key = 1;
  string val = 2;
  string sender_id = 3;
  int32 replication_factor = 4;
  int64 timestamp = 5;
}

message GetRequest {
  string key = 1;
  string sender_id = 2;
  int32 quorum_size = 3;
}

```

Listing 1: Protocol Buffer Definitions (tinykv.proto)

3.3 Modelling

- **Domain Entities:**
  - **Entry:** A Key-Value pair associated with a Timestamp (`int64_t`).
  - **Node:** A server in the cluster
  - **Hash Ring:** The Hash Ring built and maintained individually by each node
- **Domain to Infrastructure Mapping:**
  - **Entry:** Mapped to an in-memory `std::unordered_map` on specific Nodes determined by the Hash Ring.
- **Messages: (See protocol definitions in Listing 1)**

  - **Commands:** `Put` (write data, see Figure), `Get` (read data).
  - **Events:** `Ping` (heartbeat signal).

- **System State:** The state is decentralized. No single node is guaranteed to hold the global state. The System state is the cumulative of all local `kv_store` maps in the cluster.

3.4 Interaction

- **Communication:** Components communicate via Synchronous gRPC calls.
- **Interaction Pattern: Reads**
  1. **Client** sends a GET request to an arbitrary **Router Node**.
  2. **Router Node** calculates the hash and forwards the request to the corresponding **Owner Node**.
  3. **Owner Node** sends parallel RPCs to N **Replica Nodes**.
  4. **Replica Nodes** responds with their stored values.
  5. **Owner Node** finds the "winner" (Last Writer Wins) and confirms success to the **Client**.
- **Illustration:** A diagram of the Read operation data flow can be seen in Figure 3
- **Interaction Pattern: Writes**
  1. **Client** sends a request to an arbitrary **Router Node**.
  2. **Router Node** calculates the hash and forwards the request to the corresponding **Owner Node**.
  3. **Owner Node** sends parallel RPCs to N **Replica Nodes**.
  4. **Replica Nodes** acknowledge the write.
  5. **Owner Node** confirms success to the **Client** once the Write Quorum is met.

3.5 Behaviour

- **Individual Component Behavior:** Nodes store two primary states in-memory:
  - **kv_store** The local hashmap containing user data.
  - **cluster_map** a cluster_map containing the client stubs of the peer nodes and peer_last_seen which is used to determine which nodes are unreachable. A Node is determined unreachable if it misses more than one heartbeat cycle (or has not been seen in 15 seconds).
- **State Updates:**
  - Data State is updated upon receiving a valid `Put` message with a timestamp newer than the local copy.

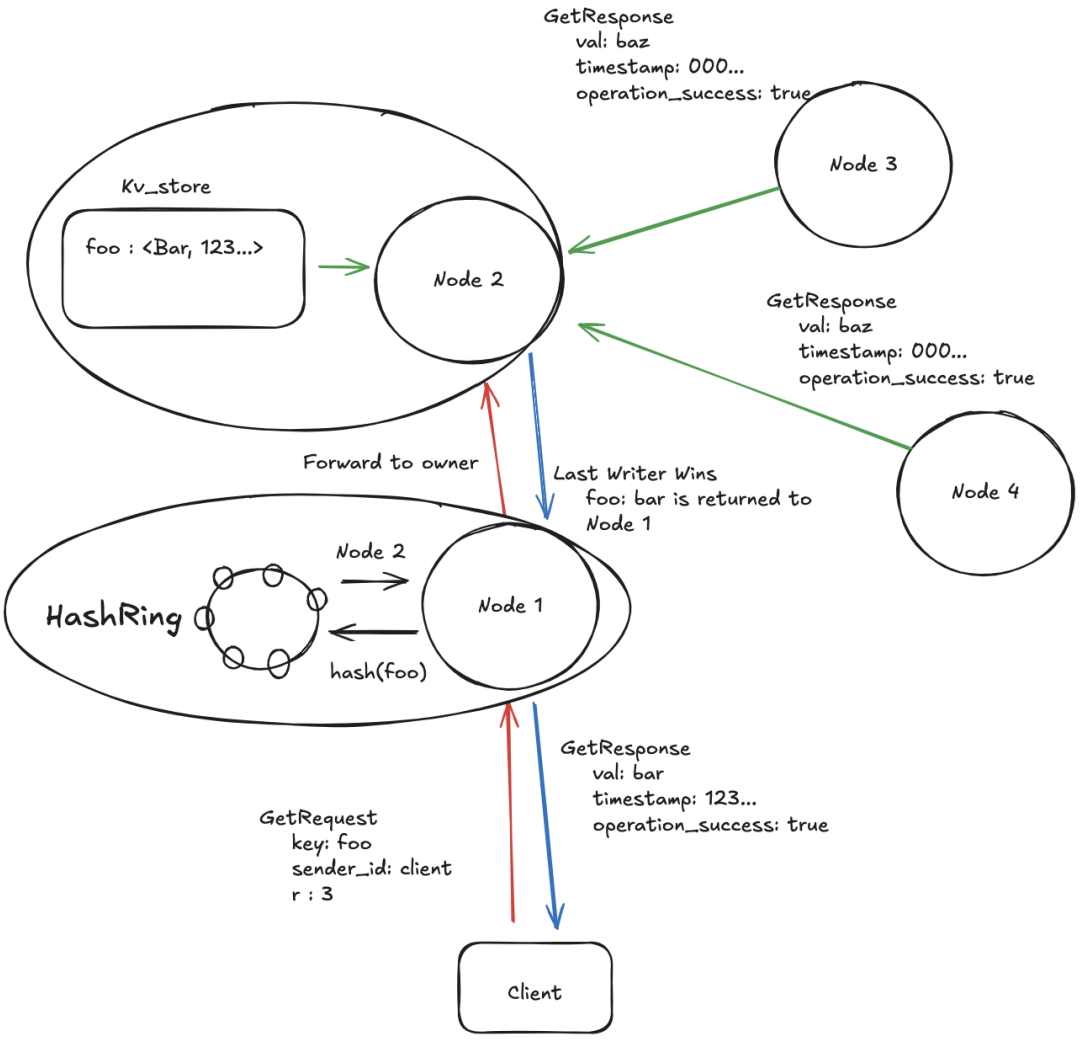

Figure 3: Read Operation data flow. Note that Client can communicate with any node in the cluster

- Cluster state is updated passively any time a node receives a request from a peer node. This means that replication requests, heartbeats, read quorums all contribute to keeping the cluster state up to date.

3.6 Data and Consistency Issues

- **Persistent Data:** Data is stored in-memory, this allows for low latency constant time read and write operations. In this way TinyKV acts like a distributed key-value cache as a total power loss would lose all data.
- **Queries:** TinyKV supports simply Put and Get queries based on Key-Value pairs. More complex queries like those found in SQL based systems are not supported.
- **Consistency Model:** The system implements Tunable Consistency through Quorums.
  - **Concurrent Writes:** The Coordinator/Owner Node will timestamp data, and any node will only overwrite data if the new timestamp is greater than the existing one ensuring LWW.
  - **Concurrent Reads:** The Coordinator queries R nodes and returns the value with the highest timestamp, ensuring the client sees the latest write if $R+W > N$.

3.7 Fault-Tolerance

- **Replication:** Every key is replicated to N unique nodes with a set default value of 3. The replication works by traversing the hash_ring to find the clockwise N (physical) neighbors to the owner.
  - **Why:** To be able to handle the simultaneous failure of $N - 1$ nodes without data loss.
- **Failure Detection:** Nodes implement an active-passive failure detector.
  - **Mechanism:** An asynchronous background thread sends Ping requests to all peer nodes every 10 seconds.
  - **Timeout:** If a node has not been seen in 15 seconds via Ping or Put/Get requests it is considered unreachable. Should an unreachable node become active again the system "self-heals" and will begin routing requests to the node again.
- **Error Handling:** A Coordinator will only abort Write request if there is not enough live nodes for replication. Otherwise it will simply go to the next in physical node on the hash_ring, until the replication factor has been met.

3.8 Availability

- **Load Balancing:** Due to the use of the **Consistent Hashing** algorithm the data is statistically guaranteed to be evenly distributed across the physical nodes in the network. This avoids any single node becoming a bottle neck for data storage. Its worth noting however that one node may still become the bottle neck if multiple clients sends requests to the same node. This could be avoiding by implementing a client facing gateway or load balances, see further discussion about this in section 9.

- **Network Partitioning:** The system remains Available in the case of a network partition. During the partition nodes in the minority can still receive writes. When the network merges again the stale data will be ignored due to Last Writer Wins for subsequent reads

4 Implementation

TinyKV is written to be highly performant with minimal overhead. Therefore the goal was to avoid bloating with web frameworks and instead rely on systems programming techniques.

- **Network Protocols:** For this project i selected **gRPC** over REST (HTTP) or raw TCP sockets.
  - **Rationale:** gRPC allows for low latency and is well suited for scalable distributed systems. It has less connection overhead compared to HTTP or raw TCP while abstracting away the complexity of serialization [2].

- **Data Representation:** Requests over the network is serialized using **Protocol Buffers (Protobuf)**.
  - **Rationale:** Compared to JSON or XML, Protobuf is a binary format and results in less memory intensive payloads and reduces required network bandwith. It also speeds up serialization and de-serialization.

- **Data Storage:** TinyKV implements a basic thread-safe In-Memory Key-Value store.
  - **Structure:** `std::unordered_map<std::string, pair<string, int64_t>>`
  - **Rationale:** Using an In-Memory Hashmap data-structure allows for constant time complexity read and writes. To support Last Write Wins (LWW), the value is stored alongside a 64 bit timestamp. The Key-Value data structure is accessed in a thread safe way with fine grained locking to maximize throughput. This is accomplished by the use of **std::mutex**.

4.1 Technological Details

- **Language Standard (C++20):** The project leverages modern C++ features to improve safety and readability.
  - **Smart Pointers:** `std::unique_ptr` and `std::make_unique` are used for managing gRPC client stubs. This prevents memory leaks by providing automatic memory deallocation.
  - **String Manipulation:** The project utilizes C++20 specific features like `std::string::ends_with` to efficiently parse address strings.
- **Containerization (Docker & Linux):** Development environment is standardized on Ubuntu 22.04 through Docker. This avoids cross-platform linking errors, as c++ libraries like gRPC act differently on MacOs and Linux.

5 Validation

Due to the importance of portability and the need for avoiding "it works on my machine" issues the validation strategy focuses on integration and end-to-end testing over unittests. Given the distributed nature of tinyKV, verifying interaction between nodes and the client was deemed more critical than testing individual components.

5.1 Automatic Testing

All automated tests are containerized, running within the same Docker network as the cluster to simulate a production-like environment.

- **Component/Unit Testing Strategy:** Due to the projects dependency on gRPC networking and peer-to-peer state tinyKV has mainly been tested manually and through end to end testing. All functionality is tested either explicitly or implicitly. Implementing proper unit tests would require significant scaffolding and changes to the build system. This is discussed further in the 8 and 9 sections.
- **Integration Testing:**
  - **Rationale:** To verify the fundamental "Put" and "Get" operations work as expected. The test spins up the cluster and a client. The client pings a node, and proceeds to execute a Put and Get request.
  - **Automation:** Implemented in `scripts/smoke_test.sh`. This script orchestrates a Dockerized client to perform a write to `Node 1` and immediately attempt a read from `Node 2` and `Node 3`.
  - **How to run:** `make test`
  - **Requirement Tested:** Ensures that replication, read forwarding and basic write and read operations work correctly
- **End-to-End Resilience Testing:**

- **Rationale:** To validate the system's behavior under failure conditions, specifically testing the Partition Tolerance (P) and Availability (A) aspects of the CAP theorem.
- **Automation:** Implemented in `scripts/resilience_test.sh`. The script writes data with a Replication Factor (RF) of 3, uses the Docker API to forcefully kill a container holding the data, and then attempts a Quorum read ($R = 2$).
- **How to run:** `make resilience`
- **Requirement Tested:** Verifies **Fault Tolerance** and **Tunable Consistency**. It proves that the system can satisfy a Read Quorum even when replicas are unreachable.

5.2 Acceptance test

- **Manual Verification:** I utilized the manual CLI entry point (`make cli`) to perform exploratory testing beyond the standard scripts.
  - **What was tested:** 1. **Log Inspection:** Logs have been monitored while performing various put and get operations to ensure that coordinator logic, replication, forwarding and quorum functionality works as expected 2. **Heartbeat Mechanics:** Verified that the logs of each node receive Heartbeat pings as expected. Each node should receive 4 pings every 10 seconds, assuming a cluster of n = 5.

6 Deployment

The deployment strategy for TinyKV prioritizes ease of use and portability. By containerizing the build and execution environments we avoid dependency conflicts across operating systems and issues with compilers or libraries.

6.1 Prerequisites

The system has been designed to run entirely in docker, meaning no local c++ compiler or libraries are needed.

- **Docker Desktop**
- **Make**
- **Git**

6.2 Installation & Execution

The project includes a `Makefile` that abstracts the complexity of Docker commands. The installation process is as follows:

1. **Clone the Repository:**

```
git clone https://github.com/emsoraffa/tinykv.git
cd tinykv
```

2. **Build the Docker Image:** This step pulls the base Ubuntu image, installs the specific gRPC/Protobuf dependencies, generates the C++ code from `.proto` definitions, and compiles the source code.

```
make build
```

3. **Start the Cluster:** This command spins up the 5-node cluster in detached mode.

```
make up
```

Expected Outcome: The terminal displays the creation of the network `tinykv-net` and the startup of containers `tinykv-node1` through `tinykv-node5`.

4. **Verify Deployment:** Run the functional smoke test to ensure nodes are communicating.

```
make test
```

Expected Outcome: A series of "OK" messages confirming that data written to one node can be read from another via replication.

6.3 Configuration

The system uses a static configuration approach to simulate a fixed cluster topology suitable for consistent hashing.

Cluster Configuration (`config/clusters.txt`): This file lists the addressable hostnames and ports of every node in the ring (e.g., `tinykv-node1:50051`). On startup, every node reads this file to build its internal Hash Ring. To add more nodes you simply add another address to the **clusters.txt** file (See figure Listing 2). You can then register another node in `docker-compose.yml` using the new address (see Listing 3).

```
1 tinykv-node1:50051
2 tinykv-node2:50052
3 tinykv-node3:50053
4 tinykv-node4:50054
5 tinykv-node5:50055
```

Listing 2: Cluster Configuration (config/clusters.txt)

```
  node6:
    image: tinykv:latest
    container_name: tinykv-node6
    command: ./build/src/tinykv_server 50056
    ports:
      - "50056:50056"
    depends_on:
      - node1
    networks:
      - tinykv-net
```

Listing 3: Adding a new node in docker-compose.yml

6.4 Containerization Engineering

The project utilizes Docker Compose to orchestrate the cluster simulation. A key engineering challenge was ensuring consistent peer-to-peer networking while keeping the simulation accessible for testing.

- **Docker Network Bridge:** a Docker bridge network named `tinykv-net`. This allows nodes to resolve each other by container name.
- **Build Isolation:** The `Dockerfile` serves as a consistent compilation environment. It installs `libgrpc++-dev` and `protobuf-compiler-grpc` on Ubuntu 22.04.
- **Test Injection:** The test scripts (like `resilience_test.sh`) are engineered to attach client containers to the `tinykv-net` network. This allows the test client to communicate with any node in the cluster.

7 User Guide

The TinyKV system interacts primarily through a Command Line Interface (CLI). This tool allows users to perform CRUD operations, inspect cluster health, and run performance benchmarks.

7.1 Accessing the CLI

To prevent network isolation issues, the CLI must be run within the Docker network context. A simplified entry point has been provided in the form of a Makefile.

```
make cli
```

This command launches a container connected to the internal network and drops the user into a `bash` shell. From here, the `tinykv_client` executable can be used to contact any node (e.g., `tinykv-node1:50051`).

```
❯ make up
docker-compose up -d
[+] Running 5/6
 ⠿ Network tinykv-net      Created
 ✔ Container tinykv-node5  Started
 ✔ Container tinykv-node4  Started
 ✔ Container tinykv-node3  Started
 ✔ Container tinykv-node1  Started
 ✔ Container tinykv-node2  Started
Waiting for cluster to stabilize...
❯ make cli
docker run -it --rm --network tinykv-net tinykv /bin/bash
root@06e4f86660d4:/app# ./build/src/tinykv_client tinykv-node1:50051 put my_key Hello 3
[Client] PutRequest success! Server is ready.
root@06e4f86660d4:/app#
```

Figure 4: Terminal Output: Put request

```
❯ make cli
docker run -it --rm --network tinykv-net tinykv /bin/bash
root@06e4f86660d4:/app# ./build/src/tinykv_client tinykv-node1:50051 put my_key Hello 3
[Client] PutRequest success! Server is ready.
root@06e4f86660d4:/app# ./build/src/tinykv_client tinykv-node1:50051 get my_key 3
Hello
root@06e4f86660d4:/app#
```

Figure 5: Terminal Output: Successful Get operation

7.2 Basic Operations

7.2.1 Writing Data (Put)

To write data, the user issues a `put` command. The system supports tunable consistency by allowing the user to specify the Replication Factor (RF).

- **Syntax:** `./tinykv_client <node> put <key> <value> [RF]`
- **Example:** Storing "Hello" with an RF of 3.

Expected Outcome: The client receives a success confirmation as seen in Figure 4. Internally, the target node coordinates the write and replicates it to 2 successor nodes.

7.2.2 Reading Data (Get)

To read data, the user issues a `get` command. The user can specify the Read Quorum (R) to enforce strong consistency.

- **Syntax:** `./tinykv_client <node> get <key> [Quorum]`
- **Example:** Reading "my_key" requiring agreement from 2 nodes.

Expected Outcome: The value is printed to `stdout` as seen in Figure 5. If the node contacted does not own the data, it will seamlessly forward the request to the correct owner using the Hash Ring.

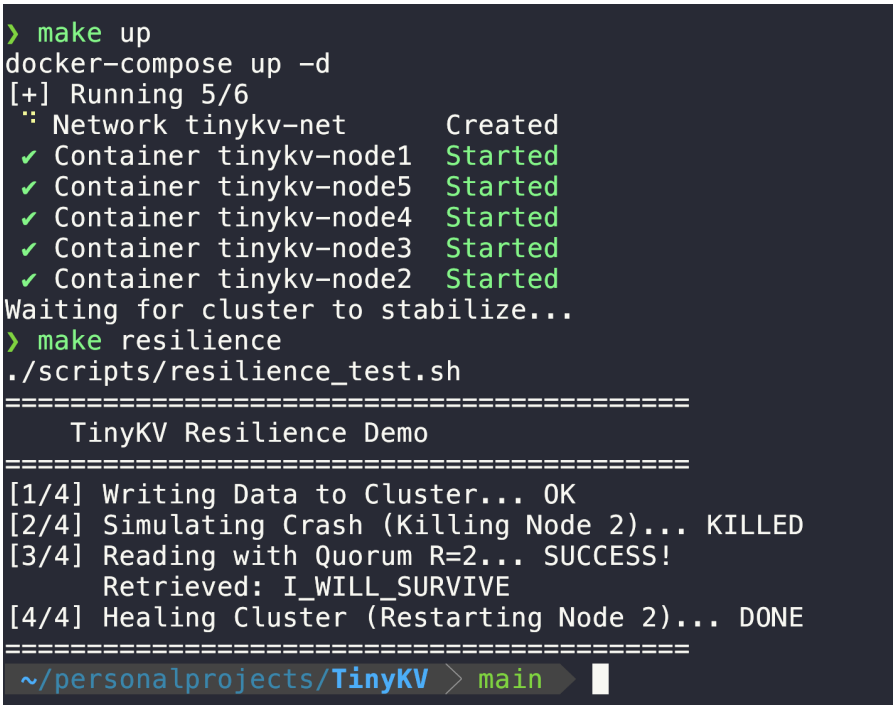

Figure 6: Terminal Output: Resilience Test

Note: TinyKV stores data in-memory, this means all data is lost when the application exits. If you exit the program after making a put request you will no longer be able to retrieve that value in a later session.

7.3 Resilience Test

The System comes with a Resilience test. This simulates the scenario where a node is killed, and ensures the Users data is not lost, Demonstrating availability and fault tolerance when R + W ¿ N

- **Command:** `make resilience`

7.4 Performance Benchmarking

The system includes a built-in benchmarking tool to measure throughput. This runs a multi-threaded C++ client that performs thousands of operations per second. Compared to a single threaded execution this is a significant speedup. For reference the single threaded execution resulted in sub 1000 Operations per second (OPS), while multi-threaded reaches upwards of 5000 OPS.

- **Command:** `make benchmark`

- **Parameters:** 10,000 requests, 50 concurrent threads, RF=3.

Expected Output: A performance report as seen in Figure 7. (Warning: Running the benchmark will flood your terminal with output)

8 Self-evaluation

8.1 Emil Rafael Solberg

TinyKV was an individual project, and as such my role encompasses the whole project. Overall the project largely meets the learning goals and motivations described in the project proposal.

The project allowed me to delve deeper and get practical experience with distributed system concepts in a low level environment. Including concepts like Consistent Hashing, Read Quorums, Failure Detection, Replication and CAP theorem to mention a few.

Implementing functionality like heartbeats, replication, consistent hashing among others, from scratch without relying on external frameworks allowed me to better understand distributed systems concepts that might previously have felt abstract.

However there is still room for improvement. While the project includes some rudimentary tests, the project could have benefited from a larger focus on test driven development (TDD). Distributed systems and concurrent systems can be hard to debug and embracing TDD early on would have made development process smoother and the end result more robust.

Another area of improvement relates to the learning goal of comparing synchronous to asynchronous network I/O. The project uses blocking gRPC requests (Synchronous), which means that this learning goal wasnt fully achieved. Comparing the performance of asynchronous and synchronous requests would have been interesting and added to the project as a whole.

9 Future works

Some interesting features and functionality to work on in the future are:

- **Gossip Protocol**: Instead of a simple heartbeat that repeats every ten seconds, a gossip protocol allows nodes to talk to each other and gossip about the state of other nodes. This would be far more scalable as as heartbeats have quadratic time complexity O(n^2) while a gossip protocol can be implemented to be logarithmic O(lg n). When the the number of nodes n grows large this is a substantial difference.

- **Asynchronous network requests**: Currently the project uses synchronous (blocking) gRPC requests, meaning a client waits until a request is finished before it sends another. By implementing asynchronous network requests for instance by using grpc::completionQueue it is possible to increase the throughput of the system

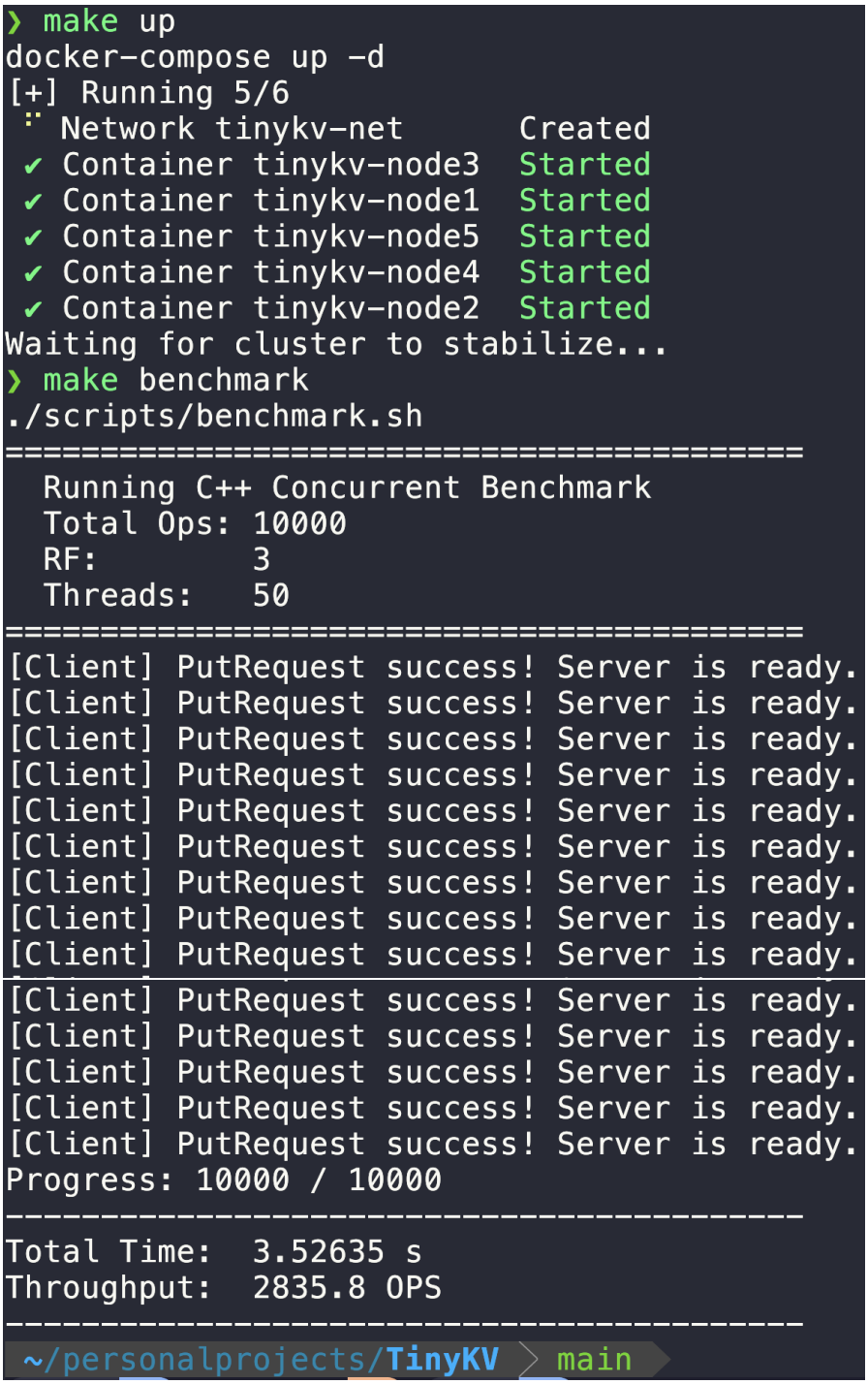

Figure 7: Terminal Output: Performance Benchmark

- **Comprehensive Test Suite:** A comprehensive test suite, including end to end testing, unittests and integration tests. This ensures the code remains robust and makes debugging easier during future development
- **DELETE Command:** A command to allow users to delete KeyValue Pairs.
- **Security:** Authentication, Authorization and encryption
- **Dynamic Membership**: TinyKV currently can support any number of nodes, and can handle a subset of nodes failing or becoming unreachable. However the number of nodes is set on startup. A Useful feature would be dynamic membership where nodes can be added on the fly, with data being dynamically redistributed.
- **Load balancing**: Since Clients can interact with any node in the cluster, there is a risk of bottlenecks in the case of multiple clients interacting with a single node. To avoid this one could implement a gateway or load balancer to ensure even network distribution across the cluster nodes.

References

[1] A.S. Tanenbaum and M.R Van Steen. *Distributed Systems - Principles and Paradigms, 4th Edition*. Maarten van Steen, 2025.

[2] The gRPC Authors. gRPC documentation. `https://grpc.io/docs/`, 2024. Accessed: 2026-02-01.