PCDP FINAL Key-Value STORE

Dabeer Ahmed

November 2024

A Journey into Distributed Systems

Building a distributed key-value store was not just a project—it was a thrilling rollercoaster ride through the intricate world of fault-tolerance, persistence, and replication. Each step came with its own challenges, testing my understanding of distributed systems concepts while pushing me to craft elegant solutions.

From designing a standalone primary replica to orchestrating seamless replication across nodes, every line of code carried the weight of making the system resilient and reliable. This report chronicles my journey, complete with moments of triumph, bugs that kept me up at night, and the deep satisfaction of bringing everything together into a fully functional distributed system.

1: Implementing the Standalone Primary Replica

To kick things off, I implemented the standalone primary replica to handle basic key-value protocol messages. The goal was simple: make the primary actor respond correctly to operations like `Insert`, `Remove`, and `Get`.

I started by setting up an internal map, `kv`, to store key-value pairs. For `Insert`, I added the key-value pair to `kv` and immediately sent an `OperationAck` message to confirm success. Similarly, `Remove` deleted the key from the map, and `Get` fetched the value for a given key.

Things seemed to be working until I ran into a problem during testing. The primary replica wasn't registering with the mediator, which was essential for proper communication. After some debugging, I realized I had overlooked sending the `mediator ! Join` message. Adding this to the `preStart` method fixed the issue, ensuring the replica registered itself automatically when started.

This step was a straightforward but essential start, laying the groundwork for handling key-value operations within the system.

2: Implementing the Secondary Replica Role

To implement the secondary replica, I focused on two core functionalities: responding to the read-only part of the key-value protocol and accepting the

replication protocol. Persistence was excluded at this stage to simplify the implementation, as guided in the specifications.

Thought Process

The secondary replica needed to:

- Respond accurately to `Get` requests, reflecting its local key-value store (`kv`).
- Process `Snapshot` messages from the primary replica for synchronization.

At this point, I used a simple approach to handle sequence numbers in `Snapshot` messages. I relied on `expectedSeq` to determine the order of updates but calculated it using the maximum of the received sequence number (`seq`) and `expectedSeq`. This approach worked for step 2 as there was no persistence or additional complexity to account for.

Implementation Details

- **Key-Value Store (`kv`):** A `Map` was used to store the replica's state, updated only via `Snapshot` messages.
- **Handling `Get` Requests:** For each `Get` request:
  - If the key existed, the corresponding value was returned using `GetResult`.
  - If the key was absent, `None` was returned.
- **Processing `Snapshot` Messages:** Each `Snapshot` contained a key, value (optional), and sequence number (`seq`). Updates were handled based on the sequence:
  - If `seq` matched `expectedSeq`, the `kv` map was updated, and a `SnapshotAck` was sent.
  - If `seq` was less than `expectedSeq`, an immediate acknowledgment was sent without updating the store.
  - If `seq` was greater than `expectedSeq`, the message was ignored.
- **Sequence Advancement:** For this step, I used `expectedSeq = max(seq, expectedSeq)` to handle sequence mismatches, ensuring the secondary replica could process incoming snapshots without crashing.

Outcome

The secondary replica successfully:

- Responded to `Get` requests, accurately reflecting its local state.
- Processed `Snapshot` messages, updated its state, and sent acknowledgments to the primary.

While this implementation worked for step 2, it later required adjustment in step 4 when persistence was introduced. At that point, I realized the use of `max(seq, expectedSeq)` could cause skipped updates in certain scenarios. I replaced it with a strict check (`seq == expectedSeq`) and incremented `expectedSeq` only after successfully processing a snapshot.

3: Implementing the Replicator

Thought Process

The skeleton code provided two essential data structures, `acks` and `toCancel`, which served as the foundation for implementing the replicator:

- `acks` **Map:** This map was used to track replication requests (`Replicate`) by associating a unique sequence number (`seq`) with the sender of the request and its details. The goal was to notify the primary of a completed replication using the `Replicated` message once a `SnapshotAck` was received.
- `toCancel` **Map:** This map stored scheduled retry tasks for each `seq`. It ensured that retries could be stopped once acknowledgments (`SnapshotAck`) were received, preventing redundant messages.

The initial implementation required integrating these tools with the `scheduler` to reliably replicate data to secondary nodes. The steps included generating a unique `seq` for each replication request, storing the sender and request details in `acks`, and scheduling retries using `toCancel`.

What Went Wrong: The Bug

In my initial implementation, I added the following line to immediately send snapshots upon receiving a `Replicate` request:

replica ! snapshot

This seemed intuitive at first—why wait for the retry mechanism? However, the skeleton code already provided a retry logic template, in which I subconsciously added a replica ! snapshot:

val cancellable = context.system.scheduler.schedule(0.milliseconds, 100.milliseconds) if (acks.contains(seq)) replica ! snapshot

The retry mechanism ensured that snapshots were periodically sent until a `SnapshotAck` was received. By including the manual `replica ! snapshot`, I inadvertently caused the following issues:

- **Duplicate Snapshots:** The secondary replica received two `Snapshot` messages for the same `seq`.
- **Test Failures:** Duplicate `SnapshotAck` responses confused the primary, which expected only one acknowledgment per request.

- **Unnecessary Complexity:** The immediate send was redundant and interfered with the scheduler's retry logic.

The Fix

Realizing the redundancy, I removed the problematic `replica ! snapshot` line and relied solely on the scheduler for retries. This adjustment ensured the following:

- Snapshots were sent at predictable intervals using the retry mechanism.
- The `acks` map remained aligned with expected `SnapshotAck` responses.
- The `toCancel` structure cleanly managed and canceled retry tasks upon acknowledgment.

The revised code became simpler and more robust:

- Each replication request added an entry to `acks`.
- A retry task was scheduled and tracked in `toCancel`.
- On receiving `SnapshotAck`, the retry was canceled, and the primary was notified via `Replicated`.

How I Used `acks` and `toCancel`

- `acks`:
  - Mapped each `seq` to the original sender and the replication request (`Replicate`).
  - Allowed me to notify the correct actor (`Replicated`) upon receiving a `SnapshotAck`.
  - Simplified tracking of acknowledgments for retries.
- `toCancel`:
  - Stored scheduled retry tasks for each `seq`.
  - Allowed retries to be canceled immediately upon acknowledgment (`SnapshotAck`).
  - Prevented redundant retry attempts and cleaned up resources effectively.

4: Implementing the Use of Persistence in the Secondary Replicas

Building on Step 2, where the secondary replica processed `Get` and `Snapshot` messages, Step 4 introduced persistence to ensure data durability and reliability, even across crashes or restarts. This required integrating a persistence mechanism into the secondary replica.

Thought Process

The objective of this step was to enhance the secondary replica's durability by persisting state updates received from the primary. Key goals included:

1. Utilizing the provided `persistence` actor to persist incoming data.
2. Ensuring `Snapshot` messages were persisted before sending `SnapshotAck` to the primary.
3. Handling persistence failures through retries until success or timeout.

A major consideration was ensuring that persistence aligned seamlessly with the sequence logic (`expectedSeq`) implemented in Step 2.

Implementation Details

1. Persistence Actor The `persistence` actor was initialized using the provided `persistenceProps`, and the replica was configured to watch the persistence actor to detect and handle failures.

2. Processing Snapshots Snapshots were processed as follows:

- **Matching Sequence (`seq == expectedSeq`):** Persist the snapshot, update the key-value store, and increment `expectedSeq`.
- **Outdated Sequence (`seq < expectedSeq`):** Send an immediate acknowledgment without persisting.
- **Future Sequence (`seq > expectedSeq`):** Ignore the snapshot.

3. Retry Mechanism Persistence operations were retried periodically using the scheduler until success or timeout. A `Cancellable` was used to track and cancel retry tasks upon completion.

4. Acknowledgment A `SnapshotAck` was sent to the primary only after persistence was successfully confirmed, ensuring reliability.

Challenges and Adjustments

- **Sequence Advancement Issue:** Initially, `expectedSeq` was updated using `max(seq, expectedSeq)`. This caused issues with out-of-order snapshots advancing the sequence incorrectly. The logic was revised to increment `expectedSeq` strictly upon successful persistence of matching sequences.

Outcome

The secondary replica achieved:

- Durability through persistent storage of snapshots.
- Reliable handling of persistence failures with retry mechanisms.
- Accurate `SnapshotAck` responses to the primary only after successful persistence.

This implementation ensured that the secondary replica could maintain its state even across crashes, setting the foundation for the primary replica's persistence and replication in later steps.

1 5: Implementing Persistence and Replication at the Primary Replica

This step marked the most challenging and transformative part of the project, where the primary replica evolved into a robust, fault-tolerant system capable of handling both persistence and replication. It required rewriting the `leader` role, introducing new mechanisms for scheduling, and carefully managing state transitions.

Refining the Primary Replica: Passing All Test Cases

Initially, my implementation of the primary replica was robust enough to handle the first three test cases effectively:

- **Case 1:** Ensuring the primary did not acknowledge updates without persistence.
- **Case 2:** Retrying persistence every 100 milliseconds until success.
- **Case 3:** Generating a failure after 1 second if persistence failed.

These cases validated my design of the persistence retry mechanism, timeout handling, and the correctness of the `scheduleForPersistence` function. The successful outcomes gave me confidence in the foundational aspects of my implementation.

Challenges with Cases 4 and 5

As I progressed to more complex scenarios in cases 4 and 5, new challenges emerged:

- **Case 4:** Ensuring the primary generated a failure if global acknowledgment failed within the timeout.

- **Case 5:** Acknowledging only after both persistence and global acknowledgment were complete.

Initially, my implementation assumed that either persistence or global acknowledgment alone was sufficient to notify the client. This shortcut led to failures in these scenarios, particularly when acknowledgments required synchronization across all replicas.

1.1 Thought Process

The primary replica had two critical objectives:

1. **Persistence:** Ensure every operation was reliably written to durable storage, with retries to handle failures.
2. **Replication:** Propagate operations to all secondary replicas, track acknowledgments, and ensure consistency before notifying clients.

This required a complete overhaul of the `leader` role to manage the dual pipelines of persistence and replication while maintaining a clear separation of concerns.

1.2 Implementation Details

1.2.1 Overhauling the Leader Role

The `leader` role was rewritten to accommodate persistence and replication:

- **`Insert` and `Remove`:** These operations updated the local key-value store, scheduled persistence, and initialized replication tracking.
- **`Persisted`:** Once persistence succeeded, the system:
  - Canceled the persistence retry task.
  - Sent replication messages to all secondaries or acknowledged the client directly if no secondaries were present.
- **`Replicated`:** This case handled acknowledgments from secondary replicas:
  - Removed the sender from the pending acknowledgment set.
  - Acknowledged the client if all replicas had confirmed receipt.

1.2.2 Scheduler and Timeout Mechanisms

The scheduler became the backbone of fault tolerance, ensuring operations were retried or timed out: To ensure robustness, I enhanced the `scheduleForPersistence` function to manage two critical tasks:

1. **Retry Task:** Periodically reattempt persistence operations until success.

2. **Failure Task:** Notify clients of an operation failure if persistence does not succeed within a specified timeout.

By encapsulating both tasks into `scheduleForPersistence`, I ensured a clean separation of concerns and minimized redundant code. This allowed retries and failure handling to work seamlessly together, improving fault tolerance and ensuring that operations were either completed successfully or gracefully failed within a bounded timeframe.

1.2.3 Using `toCancelPers`

The `toCancelPers` map was extended to manage persistence and replication tasks. Perhaps, a better name would have been `retryAndTimeoutMap`, but I used the given name. For each operation, it tracked:

- The original client who initiated the operation.
- The persistence retry task, canceled upon success.
- The failure timeout task, ensuring timely failure notifications.

Refinements and Insights

To address the issues with cases 4 and 5, I revisited and refined my implementation:

- **Tracking Global Acknowledgments:** I enhanced the tracking mechanism to ensure that acknowledgments from all secondary replicas were accounted for before notifying the client.
- **Decoupling Persistence and Replication:** I ensured that persistence and replication were treated as equally critical components of the acknowledgment process. This decoupling allowed the system to handle failures in one without impacting the other.
- **Edge Case Handling:** I rigorously tested edge cases where failures in either persistence or replication could leave the system in an inconsistent state. By addressing these, I ensured robustness.

1.3 Challenges and Fixes

- **Concurrency Issues:** Early on, race conditions caused duplicate notifications and resource leaks. Cleaning up `toCancelPers` after each operation resolved this.
- **Balancing Persistence and Replication:** Sequential handling caused delays in propagation. By decoupling persistence from replication, the system became more efficient.

- **Failure Management:** Handling cases where persistence succeeded but replication failed was particularly tricky. I tied client acknowledgments strictly to the completion of both processes to resolve this.
- **Complex Logic:** Managing everything within case blocks made debugging difficult. Refactoring key tasks into helper methods improved readability and maintainability.

1.4 Outcome

After these adjustments, cases 4 and 5 passed successfully. This process was an invaluable learning experience, highlighting the importance of testing complex interactions in distributed systems. While the initial success in basic cases indicated a solid foundation, the debugging and refinement required for more advanced scenarios revealed deeper flaws and provided a clearer understanding of how to design resilient, fault-tolerant systems.

The final implementation transformed the primary replica into a fault-tolerant, highly synchronized leader:

- **Reliable Persistence:** Every operation was persisted with robust retries and failure handling.
- **Consistent Replication:** Operations were propagated to all secondaries with acknowledgment tracking.
- **Client Reliability:** Clients were only notified of success once the system reached a consistent state.

This step was a profound learning experience, combining distributed systems principles with practical fault-tolerant design. It set the stage for the final challenge: synchronizing the initial state with newly joined replicas. .

Testing and Problem Resolution

The provided test cases played a crucial role in shaping my approach to handling newly joined replicas. By carefully analyzing these tests, I identified the required system behaviors and refined my implementation to meet those expectations.

Analyzing the Tests

- **Case 1: Starting Replication for New Replicas**
  - This test validated that the primary initiated replication for new replicas by sending `Snapshot` messages for all key-value pairs in the system.
  - It also ensured that subsequent updates and removals were correctly propagated to the new replica, with acknowledgments tracked and processed.

  - From this, I inferred the need for a method (`synchronizeKeyValues`) to iterate over the `kv` map and send `Replicate` messages to the new replicas, marked with a unique identifier (`-1`) to distinguish them as part of the initial state.

- **Case 2: Stopping Replication to Removed Replicas**
  - This test verified that the primary stopped replication to removed replicas by terminating their associated `Replicator` actors.
  - The test also ensured that no further `Snapshot` messages were sent to the removed replicas.
  - To meet this requirement, I implemented the cleanup logic for removed replicas by unregistering and stopping their corresponding `Replicator` actors using `unregisterAndStopReplicator`.

- **Case 3: Waiving Outstanding Acknowledgments for Removed Replicas**
  - This test highlighted the need to handle replicas that were removed while an operation was still awaiting acknowledgments.
  - The primary was expected to waive pending acknowledgments for the removed replicas and proceed with the operation for the remaining replicas.
  - To address this, I ensured that the `replicationCounter` was updated dynamically to remove references to replicas that were no longer part of the system.

Refining the Solution

Using the insights from the tests, I refined my implementation as follows:

- **Synchronizing New Replicas:** The `synchronizeKeyValues` function was designed to iterate over the `kv` map and send `Replicate` messages for each key-value pair to newly joined replicas. This ensured that the new replicas were fully synchronized with the current state before processing new operations.

- **Terminating Removed Replicas:** The `cleanupRemovedReplicas` function was enhanced to terminate `Replicator` actors for removed replicas, ensuring that resources were not wasted on unnecessary replication.

- **Handling Dynamic Changes:** The `replicationCounter` logic was modified to dynamically adapt to changes in the set of replicas, waiving pending acknowledgments for removed replicas and avoiding unnecessary delays in notifying clients.

Outcome

With the help of the provided test cases and a methodical approach to analyzing their requirements, I successfully implemented robust handling of newly joined and removed replicas:

- Newly joined replicas were seamlessly synchronized with the current state using the `synchronizeKeyValues` method.
- Removed replicas were efficiently terminated, preventing unnecessary resource usage.
- The system dynamically adapted to changes in the replica set, maintaining consistency and fault tolerance in all scenarios.

A Distributed Symphony Completed

What began as a simple standalone primary replica evolved into a fully functional, distributed key-value store with fault tolerance, persistence, and robust replication. Each step—from debugging errant sequence numbers to mastering asynchronous replication—taught me invaluable lessons about designing resilient systems.

The primary replica became a true leader, orchestrating synchronization like a maestro, while the secondary replicas ensured consistency and reliability. This project not only deepened my understanding of distributed systems but also highlighted Scala's elegant blend of pattern matching, object-oriented design, and functional programming. It helped me grow as a cleaner coder and sharper thinker.

For fun, I asked AI to generate answers to some challenges using the specifications. Unsurprisingly, the results were far from perfect. Distributed systems, concurrency, and fault tolerance are nuanced fields that demand more than surface-level solutions—something AI has yet to master fully. So yes, distributed systems are hard, and we engineers aren't being replaced by AI just yet!