DADTKV - A distributed transactional key-value store to manage data objects

Guilherme Pascoal 99079 | José Cutileiro 99097 | Vasco Vaz 99133

Abstract

In the realm of large-scale distributed systems, the potential benefits are manifold, yet their operation presents an array of challenges that necessitate careful consideration. These challenges encompass concurrent access management, request ordering, the resilience of constituent machines within the system, and the intricate dynamics of communication between diverse nodes. These complexities constitute a fraction of the multifaceted issues that must be diligently addressed to harness the advantages offered by distributed services. DADTKV effectively tackles these challenges by harnessing the power of the Paxos algorithm, a robust approach that empowers the development of distributed systems with inherent fault tolerance.

1. Introduction

The primary objective of this project was the conception and implementation of a distributed data store application, meticulously crafted to facilitate concurrent access to in-memory objects by multiple clients. Notably, this in-memory data spans across several distinct machines, making it imperative to maintain seamless coordination among them to ensure the smooth and harmonious operation of the system. One of the main focuses is to ensure the utmost consistency in the access behavior across different servers.

This endeavor revolves around three pivotal relationships integral to the project's success. Firstly, the interaction between clients and transaction managers, where the latter serve as custodians of data objects. Secondly, the intricate dynamics among transaction managers, governing the seamless propagation of information. Lastly, the intricate interplay among lease managers, responsible for forging consensus regarding the priority of data keys across transaction managers.

1.1. Clients and Transaction managers

The clients in this system were intentionally kept simple, primarily serving testing purposes and devoid of intricate logic. Each client establishes a robust gRPC connection with a designated transaction manager (TM). The client's core functionality revolves around the transmission of read and write requests to the TM. It is noteworthy that client operations exhibit idempotent, a crucial characteristic that substantially simplifies the intricacies of our system. Meanwhile, each transaction manager meticulously manages a storage repository containing Key-Value pairs (KVP). Upon receiving a read request, the TM promptly delivers the corresponding value for the requested key. In the case of a write request, it efficiently updates the key with the new value specified in the request.

Example: Consider the following example, accompanied by a detailed breakdown in the subsequent table. The operation Read(A) retrieves the value of key A, which, in this instance, is 0. When performing a Write(B,3), the value of B is updated to 3. Furthermore, executing Write(C,4) exhibits an interesting scenario: as C does not exist in the machine's domain, this operation results in the creation of a new KVP where C is paired with the value 4, as outlined in the table below

Before the operations		**After the operations**	
Key	**Value**	**Key**	**Value**
A	0	A	0
B	1	B	3
		C	4

(Figure 1) The state of the Transaction manager before and after the client operations

1.2. Transaction Managers

Within transaction managers, there are various design choices. But first, let's address the problems that give rise to

this array of options. The main challenge is state updating. In other words, when performing a transaction on the local state of the transaction manager, we need to somehow transmit this state to the other transaction managers. A straightforward solution would involve a general broadcast to everyone. However, what happens if a message is lost in transit to one of these servers? This would lead to one server being inconsistent compared to all the others. A foolproof solution would involve everyone who receives the broadcast sending another broadcast, and if you receive broadcasts you've already received, you ignore them and stop broadcasting. This approach ensures that the entire network, visible from my perspective and that of my peers, and recursively so, receives our state update. Nonetheless, this method incurs an unnecessary cost in terms of message exchanges, trading redundancy for coherence.

Our Transaction Managers (TMs) employ a strategy that harmonizes the strengths of two approaches for efficient message propagation. Initially, a message is dispatched to the other TMs, inquiring about the validity of the operation. Each TM independently assesses the request and responds with either 'yes' or 'no' based on its current perspective.
Following the receipt of responses from all participating TMs, the originating TM conducts a pivotal consensus check. If the majority of responses affirm the operation's validity, it proceeds to execute the operation. Subsequently, it notifies all TMs that the operation has been successfully carried out, and the corresponding values are returned in response to the client's read request. It is essential to highlight that when this message is marked as successful, each Transaction Manager (TM) proceeds to execute it locally, a crucial step in ensuring the uniformity of states across multiple TMs and upholding system-wide consistency.
In the event that the majority of responses are negative, the operation is promptly abandoned. The client is promptly notified of the request's cancellation, accompanied by the 'ABORT' message. Importantly, this scenario does not pose a significant issue, as client operations are inherently idempotent, ensuring the system's robustness and consistency.

1.3. Lease Managers

Given the potentially large number of transaction managers involved, it becomes imperative to exert control over the sequencing of client operations. This control is achieved through the application of a fundamental concept known as a 'lease.' Much like a lock, but operating at the level of a set of machines, each key-value pair on a machine is associated with a specific lease. For a transaction manager to execute client requests, it must first possess the required leases for the key-value pairs specified by the client.

To address this intricate orchestration of operations, we've introduced an additional layer within our service known as Lease Managers (LM). These Lease Managers play a pivotal role in establishing a well-defined execution order for transaction managers (TMs) concerning each key. Central to this process is the imperative need for different Lease Managers to converge on a unanimous decision regarding the order, ensuring consistent outcomes across all Lease Managers. This pivotal task of consensus is paramount.

In order to achieve this consensus, Lease Managers leverage the renowned Paxos algorithm. Paxos is the linchpin of our Distributed Asynchronous Datastore for Transactional Key-Value stores (DADTKV) system, and its role is of great importance. This choice is particularly significant because our system is built to withstand crashes and network failures, and Paxos is meticulously designed to handle such exigent scenarios with grace and resilience.

1.4. Transaction Managers and Lease Managers

As previously alluded to, the workflow involves every transaction manager forwarding a lease request to the Lease Managers. Once a unanimous consensus is achieved among the Lease Managers, they deliver the precise sequence in which the Transaction Managers (TMs) should proceed with executing the requested operations.

In strict compliance with the established order, the Transaction Managers must effectively preserve this critical information. To facilitate this, we institute individual queues for each key. Within these queues, each Transaction Manager schedules the execution of its respective request as soon as it assumes the leading position for the keys essential to the successful execution of its operation.

Example:

First Check		**Second check**	
Key	**Queue**	**Key**	**Queue**
A*	TM1 TM2 TM3	A*	TM2 TM3
B	TM2	B*	TM2
C	TM3	C	TM3

(Figure 2) In this illustrative example, it is evident that during the initial round, only TM1 is eligible to execute the operation as it has secured the foremost position in all the essential queues. In contrast, TM2 and TM3 must patiently await their turn to execute the operation.

Following the comprehensive verification by TM1, ensuring the successful completion of its assigned tasks, it

proceeds to notify the remaining TMs, as previously detailed in earlier sections. In response, the other TMs promptly remove TM1 from their respective queues. Subsequently, they perform a meticulous reassessment to determine whether the conditions are now conducive for executing their operations, or if they must continue to wait.

In this specific instance, TM2 gains the opportunity to progress since it occupies the leading position in the queues for keys A and B, affording it the privilege to act. Conversely, TM3 finds itself awaiting a bit longer as TM2 precedes it in the execution order. This iterative process persists until all operations are successfully carried out.

2. Adaptation of Paxos

Our Paxos implementation involves three actions: prepare, accept, and commit, all of which are initiated by the leader.

The prepare phase primarily seeks information about the most recent round that lease managers have accepted to recover any lost consensus due to a possible crash of a leader. Additionally, it acquires permissions to initiate a new consensus. If it obtains a majority of positive permissions, the second phase begins.

The accept phase entails sending the desired order to all lease managers. If in the meanwhile the recipients have received a newer prepare request, they reject it; otherwise, they accept our order as their own. Upon receiving all responses, the leader verifies once more for a positive majority and, if confirmed, initiates the commit phase.

The commit phase involves notifying all lease managers to inform the transaction managers about the consensus outcome.

It's important to note that leader selection is a critical aspect we need to address. To make the best use of the cyclical recurrence of consensus, we assign rounds to lease managers in sequential order. For example, the first lease manager manages the first round, the second handles the second round, and so forth.

3. Error Handling

Given our endeavor to construct a system comprised of numerous independent machines, with the exact quantity not predetermined, it becomes very important that we possess the capability to appropriately manage the potential contingencies such as machine failures or communication disruptions that might arise.

A very important note is the fact that testing these failures poses a formidable challenge due to their unpredictability. In our pursuit of assessing the resilience of our solution, we've devised a comprehensive testing system that empowers us to deliberately replicate these failures. This can be achieved by simply configuring the desired failure scenarios within our system. Generating this configuration is a straightforward process achieved by making adjustments to our configuration file. It is essential to adhere to the provided guidelines for editing this file, which are available in the original instructions or within the configuration file itself

3.1. Error handling on LM's

In error handling related to the lease manager, as previously mentioned (Section 2), we are aware that the round leader alternates among the lease managers. This mechanism, in itself, is sufficient to address a leader crash, as the leader's role is not permanent. However, in the worst-case scenario, we may need to wait for (the number of lease managers - 1) epochs to achieve a consensus.

3.2. Error handling on TM's

Our error handling algorithm for transaction managers is more intricate. To begin, it's essential to recognize that a transaction manager crashing by itself is not a trigger for error handling. The reason is that the crashed transaction manager may not have made any significant alterations, meaning there is no need to be aware of its operations, as it may not have affected the relevant state for the transaction manager responsible for error handling. However, error handling is activated when the crashed transaction manager has either attempted to modify or has already modified state related to the leases that another transaction manager requires.

In such cases, we trigger a specific mechanism. This trigger is linked to an error-handling logic, primarily involving a broadcast to all nodes we consider active. It queries whether they have received any updates from the transaction manager with highest priority concerning the leases they are waiting for. If any of the nodes have indeed received such updates, we acquire and integrate them into our state. Subsequently, we return to the normal lease verification process, as previously explained (Section 1.4). If none of the nodes have received the updates, we remove those transaction managers with priority from the queue and return to the normal lease verification process.

4. Conclusion

Our system successfully upholds a commendable balance between complexity and capabilities, safeguarding uniformity across various machines while consistently yielding accurate outcomes. This resilience extends to scenarios involving a substantial volume of machine failures and network message circulation irregularities. As such, we affirm the robustness of our solution. Moreover, our approach excels in optimizing message redundancy, a notable improvement from more resource-intensive alternatives. This efficiency significantly bolsters the effectiveness of our solution, ultimately enhancing its overall performance