Distributed Transactional Key-Value Store for Concurrent Data Management

André Jesus (110860), André Páscoa (110817), Nyckollas Brandão (110893)

Instituto Superior Técnico, Universidade de Lisboa

{andre.f.jesus, andre.pascoa, nyckollas.brandao}@tecnico.ulisboa.pt

Abstract

This paper presents DADTKV, a Distributed Transactional Key-Value Store optimized for server memory management. Designed for concurrent access across multiple machines, DADTKV ensures strict serializability of transactions that interact with data objects. The system architecture integrates client applications, transaction managers, and lease manager servers. It utilizes leases and the Paxos algorithm to achieve conflict resolution and fault tolerance. Developed in C# with gRPC, DADTKV stands as a resilient solution for distributed data management. This report delves into the design decisions and addresses potential edge-cases arising from its distributed nature, with an emphasis on fault tolerance.

1. Introduction

Distributed systems have become fundamental in modern computing, supporting a wide range of applications from cloud services to data-intensive platforms. Managing data in these systems is a multifaceted challenge, particularly in the context of concurrent data access.

The primary challenge is to maintain data consistency while allowing multiple clients to access shared data objects concurrently in a distributed system. This requires strict serializability through coordination of access for concurrent clients on different machines.

The Distributed Transactional Key-Value Store (DADTKV) project aims to meet these requirements by providing a resilient key-value store for concurrent data access.

A common approach is to use state machine replication, a key technique in distributed systems that ensures data consistency by maintaining multiple replicas that execute the same sequence of commands in the same order.

We propose an alternative approach that centers around lease-based replication to safeguard access to specific subsets of the data store while ensuring strict serializability. In this method, every replica of the data store must acquire a lease before performing a read or write operation on a subset of its data. These leases grant temporary management authority over distinct subsets of the data store without the need to lock the entire dataset. All replicas must diligently track the current lease holder and consistently distribute their updates to all other replicas.

The primary distinction between the state machine and lease-based approaches lies in their objectives. While state machine replication focuses on ensuring that all commands are delivered and executed in a specific order, lease-based replication places its focus on managing the order of accesses to specific data subsets for which lease requests have been made. Under the lease-based model, a subsequent request may execute first if it seeks access to a subset without conflicting with any preceding request.

In this report, we comprehensively document the design and implementation of the DADTKV system, which follows the lease-based replication approach, offering insights into its operation in distributed settings. We delve into the intricacies of the system's architecture, the roles of its components, and the mechanisms it employs for efficient lease management and conflict resolution.

1.1. Outline

This report is structured to guide readers through the various aspects of the DADTKV system. We begin by presenting the system's architecture, providing insights into the roles of its components and their collaboration. We then delve into the challenges and solutions related to fault tolerance. Subsequently, we explore the system's testing and validation. Finally, we conclude with the project's final remarks.

2. Architecture

The architecture of the DADTKV system is designed to provide efficient and reliable data management in a distributed environment. This section provides an overview

of the three key tiers that make up the DADTKV system: Client Applications, Transaction Managers (TMs), and Lease Manager Servers (LMs). Each tier serves a specific purpose in the overall architecture, ensuring that data objects can be accessed concurrently while maintaining strict serializability.

Client applications within the DADTKV system issue transaction requests to transaction managers, each of which possesses a replica of the data store. However, a transaction manager cannot immediately execute a transaction. Inconsistencies among replicas can arise when concurrent transactions occur, and if each transaction manager independently executes transactions without consulting the rest of the system (i.e., other transaction managers), as the transaction order may differ among replicas.

To address this, communication between transaction managers is imperative. In this specific implementation, which employs lease-based replication, a transaction manager must acquire leases for the transaction it intends to execute. Instead of directly communicating with other transaction managers, a new layer of lease managers is introduced.

To obtain a lease, a transaction manager requests a lease from all lease managers. The role of the lease manager is to aggregate lease requests from transaction managers and establish the order in which leases are to be assigned to each transaction manager. Multiple lease managers can coexist, necessitating consensus on the order of lease requests among them. To achieve consensus among lease managers, the Paxos[3] algorithm is utilized.

Once consensus is reached, all transaction managers are notified of the agreed-upon order of lease requests. Those transaction managers at the front of the lease queue can then proceed to execute their transactions locally and share updates with other transaction managers. This action signals to all transaction managers that the lease is now available, enabling the next transaction in the queue to be executed.

Upon the execution of a transaction, its outcome is transmitted by the transaction manager to the requesting client, effectively concluding the transaction cycle.

2.1. Clients

Within the DADTKV system, client applications play a crucial role in the execution of transactions. These clients use script files containing instructions for read and write transactions, sleep intervals, and system status queries for debugging purposes. To interact with the system, clients communicate with the DADTKV Client Library.

Clients form a straightforward layer of the application, primarily responsible for parsing commands from the script files and executing them. They have a limited scope of knowledge and are not concerned with lease management or other complex aspects. Their primary functions include requesting transactions or system status and patiently waiting for responses.

2.2. Transaction Managers

Transaction managers serve a pivotal role in the DADTKV system, primarily functioning as the servers that hold replicas of the data store. In DADTKV, the stored objects are simple key-value pairs known as DadInt. These pairs consist of a string key and a 32-bit integer value. This simplicity in data structure design emphasizes the distributed nature of the application rather than the intricacies of data handling. The architecture is intended to offer sufficient abstraction, making minimal assumptions about the data type stored. This design flexibility allows the system to be extended to accommodate more complex data structures in the future.

However, it's important to note that each client request doesn't typically operate on a single DadInt but aims to execute entire transactions that often involve multiple DadInt.

2.2.1 Transaction Processing Workflow

Upon receiving a client request, the transaction manager initiates the transaction processing.

Initially, the transaction manager handles client requests in a systematic manner, allowing only one transaction to be processed at a time to prevent potential data conflicts. This is achieved through a locking mechanism that controls the flow of transactions.

Subsequently, the transaction manager focus on acquiring the requisite leases. It extracts the necessary leases from the client's request and assigns a unique lease identifier. These leases are then requested from the lease managers.

The transaction manager then enters a waiting phase to confirm the successful acquisition of all required leases. It periodically checks the status of these leases until they are all in place. This waiting period ensures that transactions proceed in the correct order and minimizes the risk of premature execution.

Regarding conflict resolution, the transaction manager scans for the presence of other transaction managers that have requested leases and are waiting for their release. If such conflicts exist, the transaction manager must release the conflicting leases after executing one transaction to prevent any form of starvation or resource contention. Further insights into conflict management and lease release can be found in Section 2.2.4.

Once all necessary leases are obtained and conflicts are checked for, the transaction manager proceeds to execute the requested transaction. This involves the reading and writing of data according to the client's request.

Lastly, the transaction manager sends the results of the executed transaction and communicates them back to the requesting client, concluding the transaction cycle.

An optimization in this process involves checking whether the transaction manager already possesses leases acquired from a prior transaction, which have not yet been freed. If such leases are available, there is no need to request them again and the transaction can be executed right away.

2.2.2 Learning about Consensus

Within the DADTKV system, transaction managers play a multifaceted role as both servers for the DADTKV Client Library and learners actively participating in the consensus process. These transaction managers receive learn requests from proposers (lease managers) who have determined values for rounds in a consensus process.

Upon the reception of a learn request, transaction managers harness the Total Order Broadcast (TOB), also known as Atomic Broadcast [1], mechanism to effectively disseminate this request to their peer learners, with its focus being on receiving the requests in a well-organized and agreed-upon order. This is important for the transaction manager, as the way it handles learn requests is by placing individual lease IDs into dedicated queues, with each queue corresponding to a specific key in the datastore. The use of TOB with the consensus round number ensures that for each key, the specific order for the leases to be allocated is respected, a critical requirement for maintaining the accurate order of lease display within the queues.

2.2.3 Propagating and Receiving Updates

Once the transaction manager successfully acquires the necessary leases and completes any conflict checks, it proceeds to propagate updates to ensure data consistency across the distributed environment. This process is crucial in maintaining a synchronized state of the data store.

The propagation of updates is initiated with the transaction manager sending update requests to other transaction managers. These update requests include pertinent information such as the server's identity, the lease identifier, the write set of data to be updated, and an indication of whether the lease should be freed.

In the case where a significant number of transaction managers does not accept the update, the transaction manager needs to adopt a proactive approach. It returns a special value, typically indicating an "aborted" transaction, to prevent further inconsistencies in the data store. This ensures that conflicting data is not propagated, thus safeguarding data consistency.

Upon receiving an update request, the other transaction managers employ the FIFO Broadcast (First In First Out) mechanism to process the update requests in a specified order, similar to TOB. However, the key distinction lies in how this order is maintained. In the FIFO mechanism, the order of requests is pertinent only for each sender. This ensures that if a transaction manager sends multiple update requests before releasing the leases, the receiving transaction managers executes those update requests in order.

Importantly, there is no need for an additional layer to manage order among multiple sending transaction managers, as the consensus mechanism guarantees that only the transaction manager holding the lease has the authority to send update requests.

To execute the updates, the transaction manager holds a lock on the data store. It applies the changes to the data store by executing the read and write operations as specified in the transaction. Once the updates are successfully applied, the transaction manager marks the transaction as executed in its internal records.

2.2.4 Freeing Leases

In cases where lease conflicts arise, the responsibility falls on transaction managers to release leases once a transaction is completed. This task involves a careful examination of the lease queue for each key to identify any other transaction managers awaiting lease releases. This check occurs before transmitting an update, with a corresponding flag included in the update request to promptly signal the release of leases to others.

However, when there are no conflicting lease requests, the transaction manager retains ownership of those leases. A problem arises in such instances, as even when a future consensus value is learned, the manager lacks a mechanism to release the lease unless it receives another transaction request from a client. To mitigate this issue, when a new consensus value is learned, the transaction manager performs two checks. First, it verifies the presence of any new conflicting lease requests, and second, it checks whether the transaction associated with the manager's lease ID has already been executed. If the transaction has been completed and conflicts exist, the lease can be safely released broadcasting 'free lease requests'. If the transaction has not yet been executed, it will proceed to check for any new conflicting lease requests upon execution and release the lease accordingly through the update request as explained before.

While waiting for the acquisition of the leases of a transaction, if a timeout is reached, the current transaction manager can proceed to try a force free lease request to remove the stubborn leases preventing its own leases from being acquired. This happens in case the transaction manager holding the leases crashes or its connection with the other servers is faulty. For more information about this force free lease request, check Section 3.

2.3. Lease Managers

Lease Managers within the DADTKV system play a role in coordinating the allocation of leases, ensuring fair and efficient access to data objects. They act as both proposers and acceptors in the consensus process, orchestrating lease assignments.

2.3.1 Starting a Consensus Round

Lease Managers start by swiftly recording incoming lease requests from transaction managers, adding them to a list of pending requests and providing immediate responses.

When a lease manager accumulates a sufficient number of lease requests and can initiate a consensus round, the propose process begins. It commences by crafting the proposed value, using the accumulated requests. The proposer verifies if each request has previously been decided in prior consensus rounds; if so, it is thoughtfully omitted. Unhandled requests are added to the proposer's proposal value.

With the proposal value defined, the proposer checks if it is the leader for the consensus round. In this case, the process advances to the first phase by issuing a "prepare" message. If at least one of the promises contains an already accepted value, the proposer adopts that value as its own proposal value. If a majority of promises is achieved, the proposes advances into the second stage of the Paxos algorithm: transmitting "accept" messages.

The proposer awaits the receipt of a majority of "accepted" responses to decide the value, which is then broadcast to learners.

For both stages, if a majority of, respectively, promises and accepted responses, isn't reached, the proposer returns to the proposition stage with a higher proposal number, restarting the process.

2.3.2 Value Acceptance in Lease Management

Lease managers also serve as acceptors within the context of consensus.

For record-keeping and synchronization, acceptors maintain a comprehensive list of all consensus rounds. Each entry in this list is an object encapsulating the acceptor's state for that round, which includes the read timestamp, write timestamp, and the associated value.

When they receive a "prepare" message, they consider the proposal number. If this number is less than or equal to the read timestamp in the acceptor's state, they choose not to promise. Conversely, if the proposal number is higher, they promise to proceed.

For "accept" messages, if the proposal number matches the read timestamp, indicating that the proposer sending the "accept" is the same as the one who initiated the last promised "prepare," the lease manager accepts and adopts the proposed value, similar to a compare-and-swap operation.

2.3.3 Learning Values

In the context of consensus workflow, lease managers, like transaction managers, play the role of learners, but differently from transaction managers, do not need to be TOB (Total Order Broadcast) receivers of the consensus values. This is because lease managers do not need to apply the values in order, as they store each consensus value for every round, building a comprehensive history of all consensus values. The value for a round can be received and applied first than an earlier round as we allow "missing spots" in the list. This history is crucial for recognizing conflicts when proposing values for future consensus rounds. Refer to Section 2.3.1 to understand this proposing process.

3. Fault Tolerance

In the context of ensuring fault tolerance and liveness within the DADTKV system, we use specific forms of message broadcasting, and also address distinct aspects for Lease Managers and Transaction Managers.

3.1. Broadcasting Reliably

Most communication between processes in the system works with Uniform Reliable Broadcast (URB) [4], with the intention to broadcast the requests to all processes.

An integral aspect of our URB is the assessment of whether a majority of processes have acknowledged the request. This is called Majority-Ack URB [2], and it guarantees that if a process successfully delivers the request, all correctly functioning processes will eventually deliver it as well, adhering to the fail-silent model. To ensure the proper functioning of URB, it is essential that a majority of processes remain operational and free from crashes; there must be a majority of correctly operating processes. Furthermore, for URB to make progress, a majority of correctly functioning processes must ultimately respond.

It is thanks to the agreement property of URB that when a Transaction Manager (TM) propagates an update request, it only proceeds with the local transaction execution when it achieves delivery, indicating a majority of processes have acknowledged and eventually all correct processes will deliver it as well.

Our URB receiver logic involves not only rebroadcasting and delivering based on majority, but also respects the no duplication property, eliminating duplicated requests through the utilization of a unique message identifier (in our case, a combination of a sequence number and server ID).

In our implementation, any used Total Order Broadcast (TOB) and FIFO Broadcast (First in First Out) both work a layer above URB, using it for the broadcasting aspect itself, and mostly differ from URB in the way the messages are received (to guarantee some order).

The TOB receiver, which in the system is only used in the transaction manager to learn about consensus, arranges out-of-order requests it in an ordered list sorting them based on a message identifier that in this case was the round number. We refer to these out-of-order requests as "pending requests." As new requests arrive in the correct order, the pending requests are meticulously processed in a sequential manner, provided they are next in line.

FIFO broadcast ordering for each individual server is accomplished similarly to TOB, but by utilizing a message identifier that consists of the sequence number and the server ID (can be the same as used in the lower abstraction level, URB), and having one list of pending requests exclusive to each server.

3.2 Lease Managers

For Lease Managers, fault tolerance and consistency are guaranteed through the utilization of the Paxos algorithm. This ensures that lease assignments are maintained consistently, even in the presence of failures.

URB isn't used in the sending of prepare and accept messages by the proposer, as Paxos itself guarantees forms of retrying / rebroadcasting using new proposal numbers, as well as ways to adopt previously accepted values.

3.3 Transaction Managers

However, the fault tolerance challenge becomes more intricate when it comes to Transaction Managers. Unlike LMs, TMs do not employ any consensus algorithm. In this scenario, the system must not only guarantee consistency but also respond to issues promptly to maintain liveness. For example, when a TM crashes while holding a lease, other TMs need to detect this failure and take appropriate action to release the lease and allow the system to continue functioning smoothly.

To address this issue, we've implemented a mechanism involving a "force free lease" operation. This process is initiated when a TM takes an unusually long time to release a lease, and there is another process waiting to acquire it. In this situation, it likely indicates that the process holding the lease has crashed.

The "force free lease" operation consists of two steps:

1. **Step 1: Prepare for Force Free Lease** - Sending a "prepare for force free lease" request to other TMs, using URB to wait for a majority of acknowledges.
2. **Step 2: Execute Force Free Lease** - If majority of acknowledgments is achieved, the waiting process initiates the second step: "force free lease." It sends a "force free" request to all processes, effectively releasing the lease and enabling the system to move forward.

This two-step "force free lease" operation helps ensure fault tolerance and liveness in the absence of a consensus algorithm for Transaction Managers. It's a crucial part of maintaining the system's robustness and reliability even in the face of unexpected failures.

This implementation works in the case the process holding the lease is truly crashed as the other processes simply skip it and proceed as normal, but may fail in certain edge-cases, such as other TMs having delivered the update, but the TM requesting the force free has not, yet still delivers the force free as nobody had any say against it, only "acknowledged" that it wanted to.

For these situations, the other TMs should respond with an "ok" if they also haven't received any updates or a "free lease" from the process in question. With this, to send a force free request, the TM should require not a majority of acknowledgements but instead a majority of "ok" responses. If a majority of these processes respond with "ok", it signifies that at most a minority of processes have received (but not delivered) an update. Since this potential minority can't possibly deliver the update, as a majority is needed for updates too, by using URB; the force free request should successfully "override" the knowledge of these TMs about that update, and thus, the update isn't delivered in any TM and the transaction is effectively aborted.

For this mechanism to work, having received one type of request (update or force free), each TM should ignore (send "not ok") any future requests of the other type.

This mechanism ensures one of two: majority of processes received an update, therefore update should not be ignored and there can't be a force free; majority of processes received and accepted a prepare free, therefore force free should be sent.

For situations where a majority isn't reached, but there are enough correct processes (yet they are ignoring the requests of other type), the system should be able to employ some reliable failure detector or view synchrony, to properly remove incorrect processes from counting towards majority calculation and thus prevent this dead lock situation.

4. System Testing and Validation

4.1. System Manager and Configuration

A system manager has been developed to kick-start the system's processes. It reads a configuration file, which

holds essential details for system initialization. The system manager provides functions to initiate servers, launch clients, verify the system's status, and shut it down.

The configuration file includes data about clients, such as their IDs, the URLs of the transaction managers they connect to, and the scripts they should execute. It also contains specifics about server processes (transaction managers and lease managers), including their IDs and host/port settings. In essence, this configuration file is readily accessible to server processes, simplifying system management.

To replicate real-world scenarios, the concept of time slots has been introduced, where for each time slot a round of consensus is executed. The configuration file specifies the number of time slots and their respective duration.

Furthermore, the configuration file contains information about the state of the system, including details about processes that have crashed during each time slot and which processes suspect others. This comprehensive configuration allows for the simulation of a dynamic and realistic system environment.

4.2. Suspicions Management

At the core of suspicions management, each server process maintains three distinct lists:

1. **Processes Suspecting It:** This list contains the server processes that are suspected by the current process. The suspicions in this list arise from information in the configuration file.

2. **Processes It Suspects:** Here, the server process maintains a record of other processes that it suspects. Like the previous list, this is also influenced by the configuration file.

3. **Timeout-Based Suspicions:** In addition to suspicions derived from the configuration file, a third list is maintained to track processes suspected based on timeouts. If a server process takes too long to respond to a request, it is automatically added to this list. If the process eventually responds within the expected time frame, it is removed from the list.

It is worth noting that the suspicion mechanism is exclusively applied among server processes of the same role, ensuring that Transaction Managers do not suspect Lease Managers, and vice versa.

The suspicion logic is implemented by considering that the connection between two processes is effectively broken when one process suspects the other. In practical terms, when a process holds suspicions against another, it refrains from sending requests to the suspected process. Furthermore, if a process receives a request from another process that it holds suspicions against, it refrains from responding. This approach naturally leads to a timeout occurrence, eventually triggering further action.

Suspicions are not only vital for general system integrity but are also crucial in the leader election process for the Paxos algorithm. In this context, each process excludes the processes it suspects from the list of potential candidates for leader selection. This strategic use of suspicions ensures the reliability and efficacy of leadership transitions within the system.

5. Concluding Remarks

In summary, the DADTKV project presents an innovative approach to concurrent data access and data consistency in distributed systems.

The adoption of a lease-based replication method provides an alternative approach for concurrent data access, potentially reducing the reliance on global locks and thereby improving system efficiency.

Strict serializability, a crucial requirement for data consistency in distributed systems, is rigorously upheld through lease management and controlled data subset access.

Employing the Paxos algorithm for conflict resolution and fault tolerance enhances the system's resilience, making it robust in the face of failures.

Consistently using reliable broadcast solutions such as URB, TOB and FIFO ensures correct delivery of requests and contributes to the consistency of the system.

Efficient communication among system components, facilitated by C# and gRPC, contributes to reliable and effective data management across distributed environments.

Looking forward, there is room for further refinements and enhancements, including performance optimization, scalability, strengthened security measures, comprehensive monitoring and metrics, user-friendly documentation, and the potential for community collaboration and feedback to continue improving the system.

References

[1] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. *Information and Computation*, 118(1):158–179, 1995.

[2] R. Guerraoui and L. Rodrigues. Introduction to distributed algorithms. pages 68–72, 2004.

[3] L. Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

[4] J. Tang, M. Larrea, S. Arévalo, and E. Jiménez. Uniform reliable broadcast in anonymous distributed systems with fair lossy channels. *Computing*, 102(9):1967–1999, 2020.