DADTKV

Diogo Melita, Martim Moniz, Mateus Pinho
Universidade de Lisboa - Instituto Superior Técnico
MSc Compiter Science and Engineering

Abstract

In the field of distributed systems, ensuring concurrent access to shared data while maintaining consistency is a very challenging task. This paper presents the design and implementation of DADTKV, a distributed transactional key-value store. DADTKV uses a three-tier architecture comprising client applications, transaction managers and lease managers. Clients submit transactions, composed of read and write operations, to transaction managers, which in turn acquire leases from lease managers. The Paxos algorithm orchestrates lease assignments, regulating access to shared data and ensuring both consistency and fault tolerance. Implemented in C# and using gRPC for remote communication, DADTKV demonstrates robustness and efficiency in managing concurrent transactions across distributed environments. This project is an educational exploration that delves into distributed systems, transaction management, and fault tolerance mechanisms. While not meant for real-world use, DADTKV illustrates the practical application of theoretical concepts, deepening our understanding of distributed systems principles in a hands-on context.

1. Introduction

In the realm of modern computing, managing shared data within distributed systems is a big challenge. With organizations relying on distributed architectures for critical applications, the need to ensure concurrent access, consistency, and fault tolerance has become paramount. With this in mind we developed DADTKV, a distributed transactional key-value store to manage data objects (each one being a <key, value> pair) that reside in server memory and can be accessed concurrently by programs that execute in different machines.

1.1. Architecture

In the DADTKV system, the architectural framework consists of three tiers. The initial tier comprises client applications, the users of the DADTKV system, responsible for executing transactions on the stored data. These transactions, which are composed of read and write operations, are then routed to the second tier, comprised of transaction managers. These managers not only provide transactional access to the data but also ensure the replication of transaction updates for data durability. To manage concurrent access effectively, a system of leases is employed. These leases, determining access rights, are acquired by transaction managers from lease manager servers, the third tier. The assignment of leases to specific transaction managers is accomplished through the Paxos algorithm executed among the lease manager servers.

2. Data Model and Transactions

2.1. Data

Applications utilizing DADTKV exclusively interact with a specific object type called DadInt. Each DadInt object operates as a key-value pair, with the key represented as a string and the corresponding value as an integer.

2.2. Transactional Operations

In DADTKV, transactions involve reading and modifying DadInt objects. Clients submit transactions specifying the DadInt objects they want to access and modify. Read operations retrieve current values, while write operations modify selected DadInt object values. These operations enable complex tasks, ensuring shared data integrity.

2.3. Ensuring Strict Serializability

DADTKV ensures strict serializability. Transactions first execute read operations and then write operations

sequentially. This ordered processing prevents conflicts, maintaining consistent shared data. Adhering to strict serializability guarantees DADTKV's reliability, making it a robust solution for concurrent data access.

3. Transactional Operations in DADTKV

In the context of DADTKV, transactional operations are at the core of managing shared data and ensuring consistency across distributed environments. The system provides two fundamental methods, TxSubmit and Status, within its client library to facilitate transaction processing and monitor transaction states. The sequence of operations executed by each client is predefined by the script which is executed sequentially and in loop until the program is closed. The script also includes a "wait" instruction (for a number of milliseconds) in order to insert a waiting time between other commands.

3.1. TxSubmit

This method initiates the execution of a transaction within a transaction manager. It takes three parameters: a string identifying the client initiating the transaction, a list of strings specifying the DadInt objects to be read, and a list of DadInt objects to be modified by the transaction. The function returns a list of DadInt objects, representing the results of the transaction's read operations. In case of a transaction abortion, the return value comprises a single DadInt object with the key "abort". The client will try to send the transaction request to the first transaction manager that it is assigned to, i.e., client 1 will first try to communicate with TM1, and if it is unable to do so, it will try with the next transaction manager and so on.

Algorithm 1 Begin execution of transaction

```
1:  function TxSubmit(id, reads, writes)
2:      request ← newRequest(id)
3:      request.add(reads)
4:      request.add(writes)
5:      tm ← getTransactionManager
6:      while transaction not sent do
7:          response ← tm.Send(request)
8:          if response ≠ null then return DadIntsRead
9:          else
10:             tm ← get next Transactio Manager
```

3.2. Status

This method is used by the clients to request the status of every node in the system. The client will send a status request to every node of the system. When a node receives the Status Request, it will display on its console its state.

Algorithm 2 Status Request

```
1:  function Status
2:      request ← newStatusRequest()
3:      for all Nodes do
4:          send(request)
```

4. Leases

Transaction managers employ leases to secure exclusive access to a specific set of DadInts. Once acquired, a lease remains in the possession of the requesting transaction manager indefinitely, until it identifies a conflicting lease allocated to another transaction manager. Leases enable a transaction manager to execute multiple transactions that necessitate the same lease without the need for consensus. However, this advantage is only significant if transactions managed by different entities are expected to access distinct data sets, reducing the frequency of lease acquisitions.

5. Transaction Managers

Transaction managers in the DADTKV are a set of nodes of the system that play a pivotal role in ensuring the integrity and consistency of shared data across distributed environments. Responsible for processing and coordinating transactions initiated by clients, these managers orchestrate a series of operations to guarantee strict serializability and fault tolerance. All transaction managers store a full copy of the set of DadInt in the system.

5.1. Transaction Execution Workflow

Upon receiving a transaction request from the client, the transaction manager executes a predefined workflow in order to process the transaction request.

Lease Checking: At the moment of receiving a transaction request, the transaction manager will first check if it holds the appropriate leases for the DadInts included in the request. If it does, it will jump to the Transaction Processing step, otherwise it will need to request the leases from Lease Managers.

Acquiring Leases: If a transaction Manager does not have the appropriate leases to execute the transaction received, it will send a Lease Request to the Lease Managers. However, the response from Lease Managers will arrive asynchronously, at the beginning of the next slot.

Transaction Processing: Once a transaction manager has obtained all the necessary leases for the received transaction, it proceeds with its execution. Initially, it processes the read operations to prevent conflicts, ensuring the consistency of shared data. Subsequently, the transaction manager executes the write operations, as detailed in the following explanation.

5.2. Processing Write Operations

Following the completion of read operations, Transaction Managers proceed to handle write operations. Immediate processing isn't feasible due to the necessity of propagating these changes to other Transaction Managers to maintain stable and consistent data. To ensure data integrity, we've implemented a robust approach - the Two-Phase-Commit protocol, augmented with a majority consensus.

In the initial phase, the initiating Transaction Manager dispatches a Prepare message to other Transaction Managers, awaiting their responses. The transaction can only proceed if it receives affirmative responses from a majority. Should this condition not be met, the transaction is aborted and the client will receive a single DadInt with the key "abort".

Upon a successful preparation phase, the transaction enters the Commit Step. Here, the initiating Transaction Manager communicates the write operations to be executed by other Transaction Managers. These managers promptly execute the transactions, and the originating Transaction Manager processes them subsequently.

This methodology, employing the Two-Phase-Commit protocol coupled with a majority consensus, ensures that transactions are executed only when a majority of Transaction Managers are available. Even if a subset of managers does not receive the Commit request, at least one Transaction Manager will execute the transaction. Additionally, by maintaining a comprehensive write log of operations, the system conducts a synchronization check at the onset of each slot, ensuring seamless coordination across all Transaction Managers.

Algorithm 3 Transaction Processing

```
1: function ExecuteTransaction(transaction)
2:     for all Transaction Managers do
3:         send(PrepareRequest)
4:     if #PrepareResponses > #TMs/2 + 1 then
5:         send(CommitRequest(transaction.Writes))
6:         processWriteOperations(transaction.Writes)
```

5.3. Slot Preparation

At the beginning of each time slot, every Transaction Manager undergoes a series of updates and checks to ensure system integrity. Initially, the manager updates its status to either "crashed" or "normal" based on the configuration file and adjusts its list of crashed processes accordingly.

Subsequently, the Transaction Manager attempts to update its transaction log status. During this phase, it requests written operations logs from other managers. At least one manager is guaranteed to have an updated log which will be the largest one, and in this case, the requesting manager synchronizes with the one possessing the largest log. If the manager's log is outdated, it relinquishes all held leases and aligns its local log with the most extensive and common log received.

Finally, the Transaction Manager contacts the Lease Managers for a State Update to obtain newly assigned leases. After acquiring these leases, the manager checks for conflicting leases and reviews pending transactions. If the manager possesses the necessary leases to execute a transaction and there are no conflicting leases within the same time slot, the transaction is processed as mentioned above.

Algorithm 4 Slot Preparation

```
1: function PrepareSlot
2:     Tm.state ← Config.GetState(TM)[slotID]
3:     for all TransactionManagers do
4:         if TM.state() == crashed then
5:             crashedProcesses.Add(TM)
6:     UpdateLog()
7:     AskLeaseManagersForUpdate()
8: function UpdateLog
9:     logs ← newList()
10:    request ← newUpdateLogRequest()
11:    for all not crashed and not suspected TMs do
12:        response ← TM.sendLog(request)
13:        logs.Add(response)
14:    GetMajorityResponses()
15:    largestLogSize ← LargestCommunLogSize()
16:    if currentLog.size() < largestLogSize then
17:        UpdateCurrentLog()
18: function AskLeaseManagersForUpdate
19:    request ← newUpdateStautsRequest()
20:    for all LeaseManagers do
21:        sendUpdateStatusRequest(request)
22:    CheckConflictingLeases(UpdateStatusResponse)
23:    for all Pending Transactions do
24:        if All leases and No Lease Conflicts then
25:            ExecuteTransaction(transaction)
```

5.4. Implementation Decisions

In our implementation, we opted for the Two-Phase Commit protocol over gossip for several compelling reasons. Firstly, our system demands strong consistency and transactional guarantees, even if it means accommodating potential blocking and heightened latency associated with synchronous coordination. Two-Phase Commit ensures the atomicity of transactions, a critical aspect that either commits all Transaction Managers' transactions or none at all. This property is paramount to maintaining the system's correctness and preventing inconsistencies.

To balance the need for consistency with client expectations of low latencies and high availability, we devised a modified version of the protocol using majorities. When our processes communicate with clients, they wait for a majority of "Ok" responses from Prepare Requests before proceeding with transactions. This method grants our managers the confidence to proceed, as a majority commitment ensures that the transactions will be executed.

Moreover, at the beginning of each new slot, we conduct a preparation phase to guarantee that every manager in DADTKV is up to date. This step is crucial because failures might occur during the execution of the Two-Phase Commit protocol, potentially causing a manager to miss the commit message containing the corresponding write operations. By performing this preparation, we ensure consistency throughout our system, verifying that every manager is impeccably updated.

6. Lease Managers

The role of lease managers is to order Lease Requests and assign leases to transaction managers, using the Paxos algorithm.

6.1. Slot Preparation

At the start of every time slot, each Lease Manager updates its status based on the configuration file, for the other Lease Managers checks which are crashed and stores that information and, finally, it initiates a new Paxos epoch. However, only the Leader has the authority to commence the new Paxos epoch by sending a "prepare" message to the other Lease Managers, who await this signal.

6.2. Paxos Implementation

A Paxos epoch commences at the start of a new slot and is triggered by the Leader, only if no existing Paxos epoch is running and no value has been decided yet. The Leader Lease Manager initiates the process by sending a prepare request to the appropriate Lease Managers and awaits a majority of promise replies. Lease Managers responding to the prepare request include their read timestamp and leased information.

Once a majority of responses is gathered, the leader examines all received replies to identify the most recent one, determined by the read timestamp. If a more recent Lease Manager is found, it assumes the role of the leader. Subsequently, the Paxos slot progresses to the next step, where the leader proposes lease values. It sends an accept request to all appropriate Lease Managers and awaits the decision made by learners.

Upon receiving an accept request, Lease Managers reply with their write timestamp and leases in a decide request. Learners then decide on a value by selecting the most common response received and relay this information. Upon completion of a Paxos slot, each Lease Manager stores the decided leases and provides them to Transaction Managers upon request.

Algorithm 5 Paxos

```
function PAXOS
    if Paxos Leader then
        sendPrepareResquest()
    for all PromiseResponses do
        if response.ReadTimeStamp > LeaderID then
            WaitPaxos()
    Leader.SendAccept(value)
    WaitForLearnersToDecide()
```

7. Conclusion

In conclusion, our endeavor to design and implement the DADTKV system has been a significant exploration into the complexities of distributed systems, transaction management, and fault tolerance mechanisms. Through meticulous planning and rigorous implementation, we have achieved the successful realization of a distributed transactional key-value store that upholds strict serializability, consistency, and fault tolerance.

Our work demonstrates the vital role of transaction managers and lease managers in orchestrating concurrent access to shared data while ensuring data integrity. By opting for the Two-Phase Commit protocol, enhanced with majorities, over gossip-based solutions, we have prioritized strong consistency, transactional guarantees, and system stability. The use of leases, managed through the Paxos algorithm, has enabled us to regulate access to shared data effectively, ensuring non-conflicting transactions can proceed concurrently.

Throughout this project, we have encountered and overcome various challenges, leading to valuable insights into

the nuances of distributed systems. Our implementation decisions, guided by theoretical concepts, have translated into a robust and efficient system capable of managing concurrent transactions across distributed environments.

References

[1] G. Coulouris. *Distributed Systems: Concepts and Design*. Pearson Custom Publishing, 2012.

[2] U. Joshi. *Patterns of Distributed Systems*. Addison-Wesley Professional, 2023.