# DADTKV Project Report

**Design and Implementation of Distributed Applications 2023-2024**
**Group 06**

Manuel Albino
Instituto Superior Técnico,
Universidade de Lisboa
manuel.albino@tecnico.ulisboa.pt

Guilherme Lima
Instituto Superior Técnico,
Universidade de Lisboa
guilhermegutierrez727@tecnico.ulisboa.pt

Guilherme Batalheiro
Instituto Superior Técnico,
Universidade de Lisboa
guilherme.batalheiro@tecnico.ulisboa.pt

## Abstract

*DADTKV is a Distributed Transactional Key-Value Store designed to efficiently manage data objects, each represented as a <key, value>pair, residing in server memory.*

*This project focuses on Consistency and Fault tolerance in every process involved, including Clients, Transaction Managers, and Lease Managers.*

*We will be describing the features of these three types of processes and how they communicate with each other, along with problems and mistakes made during the development.*

## 1. Introduction

Managing consistency, fault-tolerance, and availability within a single **Storage System** is a prominent and widely discussed topic in the modern era.

The Design and Implementation of Distributed Applications **(DADTKV)** project, undertaken during the academic year 2023-2024, seeks to address this critical challenge by introducing a distributed transactional key-value store. This system, referred to as DADTKV, is designed to efficiently manage data objects in a distributed environment. Each data object takes the form of a <key, value>pair and resides in the memory of Transaction Managers, making it accessible to Clients executing on various machines simultaneously.

Throughout this report, we will explore the layers of DADTKV, starting with the architecture. This includes the client that interacts with the **Transaction Managers**, these are responsible for maintaining a replication of the memory across them and answering the **Clients** transactions, and **Lease Manager** servers regulating concurrent access of **Transaction Managers** to Leases.

We will also discuss the use of leases, which are pivotal in managing data access, and how they are obtained and allocated among **Transaction Managers** through a consensus algorithm.

To simplify the project scope, DADTKV is designed to manage a specific type of data object known as *DadInt*.

In summary, this report provides an in-depth exploration of the DADTKV project, highlighting its significance. It aims to elucidate the project's design, implementation, and the challenges associated with achieving consistency, and fault tolerance in a distributed storage system.

Through the following sections, readers will gain a comprehensive understanding of the project's complexities, its components, and the methodologies employed to achieve its objectives.

## 2. Overall Architecture

Each process has these classes in order to help them implement their code:

- **DebugClass**: A class that implements logs to make debug easier, enable each Log in the Program.cs of the correspondent Class.
- **Program**: Starts the Process Class itself and passes the argument to them, utilizing the FlagReader Class.
- **FlagReader**: Carries the arguments that each process needs to function correctly.

# 3. Initializer

The **Initializer** is the class that parses the configuration file, creates each process node and sends the respective arguments to each one of them.

## 3.1. Problems

The parsing of the Rounds Down argument has an error that makes the Transaction Managers and Lease Managers not be the same as in the configuration file.

# 4. Client

## 4.1. Client Architecture

The Client's purpose is to execute a list of commands given through a script. To make this work we divided it into a set of classes:

- **Client's class**: Where the attributes and the logic of the Client are.
- **Client Service**: Where the Client communicates with the Transaction Manager.
- **Script Parser**: Where the Client parses the script file into commands.
- **Command Structure**: This Command Structure Class implements the multiple commands that the Client can execute.

## 4.2. Client Library

The Client library consists of 3 commands: ***SubmitTransaction***, ***Status*** and ***Wait***.

In the ***SubmitTransaction*** command, a Client submits a set of ***DADInts*** he wants to read and a set he wants to write to the Transaction Manager. Each ***DADInt*** object represents a key-value pair, with the key encoded as a string and the value as an integer.

In the ***Status*** command, a Client sends to all the Transaction Managers a request for them to print their current status, either *Crashed* or *Normal*.

In the ***Wait*** command, a Client awaits the amount of milliseconds specified in the command.

## 4.3. Client Communication

The Client communicates to the Transaction Managers through *gRPC* channels, these channels are established via an address that is passed to the Client by an argument flag. The communication is **asynchronous** so that the Client is capable of sending multiple requests simultaneously without having to wait for the Transaction Manager to respond to his requests.

We also implemented a **LoadBalancer** that consists of randomizing the Transaction Manager that a Client connects to for each communication.

# 5. Transaction Manager

## 5.1. Multiple Client Support

Each **Transaction Manager** will be connected to one or more clients and will receive each transaction asynchronously, which means each client will be able to submit every operation they possess at once.

The Transaction Manager will receive every operation and look for the **Leases** it requires. In the case that it already possesses all Leases necessary for the operation it will **immediately** execute it. For the operations that are missing Leases, however, they will have to wait until these are released and assigned to the TM, before they can be executed.

This way, **Clients** do not have to wait for Leases to be given to the respective Transaction Manager on operations where the TM already has all Leases available. In **earlier** stages of the project, every Client received a reply to each operation one at a time, which could cause the situation described.

As a short example, let's imagine the **Transaction Manager** connected to the **Client** already has the Lease *"A"*, and the Client wants to read the value of the key *"C"* and afterward read the value of *"A"*.

In our **previous version** of the project, the Client would need to wait for the Transaction Manager to request and receive the Lease *"C"*, before it could read the value of *"A"*. With our **improved version**, we are able to receive both of these read operations and evaluate which can be executed right away. In that case, the reading *"A"* will be executed without the need to wait for the reading *"C"*.

## 5.2. Lease Requests

When receiving operations, the **Transaction Manager** will have to evaluate which **Leases** it is missing to execute

them individually, if they do not hold the necessary Leases, they will have to request them from the Lease Managers and await a response. This response will contain the ordering of all the leases and the TM assigned to each one, in that *Round*.

When this *Lease List* is sent by the majority of the **Lease Managers**, it is going to be authorized and analyzed to make sure each Transaction Manager receives the necessary Leases without any conflicts.

#### 5.2.1 Transaction Epoch

A ***Transaction Epoch*** is how we separated transactions between *Time Slots*.

This Transaction Epoch has a batch of transactions that are requested by the clients, and when the **Paxos** runs, the Leases corresponding to each Transaction are given to his Transaction Manager.

If a Transaction is received by a Transaction Manager while the Paxos is still running and he hasn't learned the order of the Leases yet, this Transaction will wait until the end of the Paxos. The Transaction is then pushed to the next Time Slot and Epoch.

#### 5.2.2 Lease Release and Propagation

After analyzing the *Lease List* received by the Lease Managers, each **Transaction Manager** will execute two distinct processes, the *LookBack*, and the *LookAhead*.

This *Lease List* will have the **order** of the Leases that each Transaction Manager will have so that every TM knows who holds what Leases and when, making sure to respect the order received.

The *LookBack* algorithm we implemented, analyzed the *Lease List* to see what Leases we require that are going to be used by another **Transaction Manager**. The TM will wait until it eventually has all missing Leases, and only after that will it release them, this Lease will be released by the *LookAhead* algorithm.

Similar to the *LookBack* algorithm, *LookAhead* goes through the *Lease List* and checks if any **Transaction Manager** will require the Leases we hold after our transactions are done, that way it knows what leases to **release** and **send**.

### 5.3. Memory Propagation

Finding a way to broadcast the memory of every **Transaction Manager** was a challenge, considering the **Consistency** in the values returned to the Clients.

Every **Paxos** round will gather a batch of transactions, or as we call it, a *Transaction Epoch*. This batch will be looked over to see the last write operations for each Lease, values which will be decided for the Memory **Consensus** within the Transaction Managers.

Even though the TMs are aware of who is the last to use a specific Lease, they do not know if it is going to be a read or a write operation, or if the value of the Memory is going to be changed.

That's where the main issue surges, since the Transaction Manager that executes the last write operation of the *Transaction Epoch* will have to *broadcast* this information to every other TM. For this *broadcast* we implemented a variation of **Uniform Reliable Broadcast**.

After the *LookAhead*, the Transaction Manager will execute the **Fission Algorithm**.

#### 5.3.1 Uniform Reliable Broadcast

**Uniform Reliable Broadcast** (URB) is an algorithm used to ensure that every correct process delivers the same set of messages in the same order. This algorithm, however, is used along with a **Perfect Failure Detector**, one that we do not have in this project, hence why this is a variation of URB.

Once a Transaction Manager finishes every write operation, it will propagate the changes made to every TM. The first time these TMs receive a message of update in the memory, they will propagate this to every TM as well. To avoid a chain reaction that leads to an **infinite loop**, these will only do this once per update.

When the TMs receive the message regarding the update from the **majority of the Transaction Managers** (instead of the number of live processes that we do not have access to), they will execute the memory update in their own **Server** and the Propagation of the Memory will be complete.

This however is not going to be always the case since in the case of a **Crashed Transaction Manager**, we do not know when to complete the Update to the Memory, making this only viable with all TMs working.

#### 5.3.2 Fission Algorithm

Inspired by the splitting of atoms in Nuclear Fission, we implemented an algorithm that would warn all **Transaction Managers** about what leases can be released.

After the *LookAhead* algorithm we will see what Leases need to be sent, and to whom. These Leases will be propagated in the **Fission algorithm** to every other TM, with both the name of the Lease and the TM it is headed to.

When a Transaction Manager receives this message that a Lease is released and wanted, it will verify if that same Lease is for itself. If it is, the propagation ends and it will add that Lease for itself. In the case that it isn't, the Transaction Manager will continue to **propagate** to the other Transaction Managers until it reaches the **desired one**.

This algorithm is an **efficient way** to release Leases that are no longer needed and wanted by other Transaction Man-

agers, and also **tolerates faults** where a TM suspects another. Let's say, for example, that *TM1* suspects *TM2* has crashed and will not communicate with it, even though *TM2* requires a Lease from *TM1*.

In this case, by using the **Fission Algorithm**, and with the help of the other Transaction Managers, that Lease will arrive at *TM2*, considering that the other TMs do not suspect *TM2*.

In our project, there is a flaw in the implementation of this algorithm. When we receive a lease, we need to ensure that we aren't receiving it again. Currently, we achieve this by incrementing a variable and checking if the incoming request id is larger. However, this poses a problem because the Transaction Managers (TM) don't share this integer. To address this issue, we could be maintaining a list of integers, where each element corresponds to a specific TM. This way, we can guarantee that we won't receive the same lease twice.

# 6. Lease Manager

## 6.1. Lease Requests

Every Lease Manager is linked to each Transaction Manager through a one-to-many connection. These connections are designed to await requests from the Transaction Manager for one or more leases, which will then be recorded in a list of requested leases within a specific **Paxos** round.

Each Lease Manager has a timer that triggers the **Paxos** algorithm periodically within every time slot.

## 6.2. Paxos Elements

### 6.2.1 Proposer

When this algorithm starts, each Lease Manager checks if it is a **Proposer** and also verifies if is buffer is not empty. To determine if it is a **Proposer**, each Lease Manager is provided with a list of Lease Manager IDs as an argument at the beginning. In this list, the Lease Manager at index 0 always becomes a **Proposer**. If a Lease Manager suspects the one preceding it in the list, it will also assume the role of a **Proposer**. A proposer triggers the consensus process by suggesting a value, such as a list indicating the order in which Transaction Managers hold leases, to be collectively endorsed within the system.

### 6.2.2 Acceptor & Learners

Every Lease Manager that is not a **Proposer** becomes an **Acceptor**. An acceptor in the Paxos algorithm is responsible for acknowledging or rejecting proposals from proposers. All Transaction Managers are **Learners**. Learners are responsible for receiving and adopting the agreed-upon value.

## 6.3. Paxos Algorithm

At the start, the **Proposers** initiate by sending 'prepare' messages to each **Acceptor** along with a read timestamp. Upon receiving a 'prepare', an **Acceptor** verifies if it has already committed to a larger read timestamp for another **Proposer**. If not, it records this number and responds to the **Proposers** accordingly. Otherwise, it issues a negative response, signifying an inability to make a commitment. Additionally, it checks if it has previously accepted any values. If so, it includes that value in the response but only if he 'promised'. The **Proposers** await a majority of successful responses, and if any of these responses contain a value, it indicates that the value has already been accepted.

Once the **Proposers** obtain a majority of **Promises**, they proceed to send 'Accept' requests to all **Acceptors** along with the value to be accepted and the write timestamp. Upon receiving this request, an **Acceptor** verifies if the read timestamp matches the value it possesses. If so, it stores the value sent by the **Proposer** and responds by confirming acceptance. And after this the **Acceptors** will send learn request to the Transaction Managers with the paxos round number.

The **Proposers** continue to run this algorithm until they successfully receive a majority of acceptances.

## 6.4. Algorithm Problems

This algorithm relies on a majority of **Acceptors** for its operation, excluding the **Proposers**. This issue could be addressed by having the **Proposers** also send learn requests to the Transaction Managers. Additionally, the implementation does not account for potential live locks. Another concern with this approach is that the **Acceptors** always wait for a constant value of majority. A potential solution could involve implementing a mechanism for the **Acceptors** to decrement this value when they suspect another **Acceptor**.

# 7. Conclusions

In this report, we have addressed every feature, problem, and solution attributed to DADTKV, a storage system with a strong emphasis on consistency, fault tolerance, and Consensus between the processes involved.

We believe DADTKV is an excellent option for a storage system, and despite all the issues involved with the availability of the project, it is a capable, simple but powerful system.